Cyclic Register Pressure and Allocation for Modulo Scheduled Loops

Sid Touati, Christine Eisenbeis

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Cyclic Register Pressure and Allocation for Modulo Scheduled Loops

Sid-Ahmed-Ali TOUATI — Christine EISENBEIS

N° 4442

April 18, 2002

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4442--FR+ENG

INRIA
ROCQUENCOURT

Cyclic Register Pressure and Allocation for Modulo Scheduled Loops

Sid-Ahmed-Ali TOUATI, Christine EISENBEIS

Thème 1 — Réseaux et systèmes
Projet A3

Rapport de recherche n° 4442 — April 18, 2002 — 127 pages

Abstract: In a previous work[TT00, Tou01d, Tou01e, Tou01b], we have introduced the notion of register saturation in directed acyclic graphs (basic blocks) which is the maximal number of registers needed to complete a computation in a multiple issue processor. In this report, we extend our work to the cyclic case : given a data dependence graph of a simple loop, the cyclic register saturation is the maximal number of registers needed by any modulo schedule of this DDG. If the register saturation is lower than the number of available registers R, then we can build a software pipelining schedule without including the registers constraints : we are sure that any schedule does not need more than R registers, and so no spill code has to be generated. If not, we add some arcs in the DDG such that any modulo schedule would not require more than R registers, while minimizing the critical circuit. Next, we introduce and study the dual notion : the cyclic register sufficiency is the minimal number of registers needed by any modulo schedule of the DDG. If this factor is greater than R, spill code cannot be avoided. Finally, we shall show how to construct a cyclic register allocation in the data dependence graph independently from any software pipelined schedule. We insert some anti-dependences into the original DDG to express the register reuse relations between statements, such that the number of consumed registers does not exceed R under a fixed execution rate (initiation interval).

Key-words: register constraints, register requirement, register need, register pressure, cyclic register saturation, cyclic register sufficiency, software pipelining, register allocation, integer linear programming

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Consommation et Allocation Cycliques de Registres dans les Boucles Simples

Résumé : Dans un travail précédent[TT00, Tou01d, Tou01e, Tou01b], nous avons défini la notion de saturation en registres dans les graphes de dépendances de données acycliques (blocs de base), qui est le nombre maximal du besoin en registres pour tout ordonnancement acyclique dans un processeur à parallélisme d'instructions. Dans ce rapport, nous étendons cette notion au cas cyclique : ayant un graphe de dépendances de données d'une boucle simple, la saturation cyclique en registres est le nombre maximal de registres nécessaire pour tout ordonnancement pipeliné de cette boucle. Si cette saturation est en dessous du nombre de registres disponibles R, alors nous pouvons construire un pipeline logiciel sans inclure aucune contrainte sur les registres : nous garantissons que n'importe quel ordonnancement cyclique n'utilisera pas plus que R registres. Par conséquent aucun code de vidage n'est nécessaire. Dans le cas contraire, nous ajoutons des arcs dans le graphe original pour assurer que la saturation ne dépasse pas le nombre de registres disponibles, tout en garantissant un circuit critique minimal. Nous introduisons et étudions également la notion duale : la suffisance cyclique en registres est le nombre minimal de registres requis pour obtenir un calcul valide. Si cet nombre est supérieur à R, alors l'introduction du code de vidage est nécessaire. Finalement, nous montrons aussi comment construire une allocation cyclique de registres dans un graphe de dépendances sans ordonnancement à priori. Nous insérons des arcs d'antidépendances entre les instructions de la boucle pour représenter les relations de réutilisations des registres. Les contraintes à vérifier sont, d'une part, de ne pas consommer plus de registres que ceux disponibles, et d'autre part, de minimiser le circuit critique dans le graphe engendré.

Mots-clés : contraintes de registres, besoin en registres, consommation en registres, saturation cyclique en registres, suffisance cyclique en registres, pipeline logiciel, allocation de registres, programmation linéaire en nombres entiers

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The problem of minimizing the register requirement for vertical straight-line code[1] (single issue processors) is an old problem proven NP-complete in [Set75] for general DAGs. On such machines, the latencies of the operations were assumed unit, and hence the code optimization techniques were focusing on reducing the number of executed operations. An optimal register allocation can be either finding an order for the operations with a minimum number of registers, or fixing this order and minimizing the amount of spill code introduced. In the case where the DDG is a tree (arithmetic expression for instance), the optimal register allocation can be computed in polynomial complexity. The problem of minimizing the number of registers needed to evaluate an expression tree without spills was first resolved by Nakata [Nak67] and Redziejowski [Red69]. Sethi and Ullman extended that result in [SU70] to minimize the amount of spill code needed to evaluate an expression tree, given a fixed number of registers.

In pipelined architectures, some operations can have a delay but the code is still linear. By assuming that there is no other explicit parallelism in the generated code except in the pipeline, some related work treats the register allocation problem under this assumption [KPF95, PF91, Kes98]. Finding an optimal register allocation for expression trees with a delayed load of 1 has been proved a polynomial problem in [PF91, KPF95]. For delays larger than 1, the problem still remain difficult.

With the introduction of multiple issue processors (VLIW and superscalar processors), the semantic of the operations changed : their latencies became not unit and visible to the compilers. Since the operations can be executed in parallel, the code optimization techniques migrated towards minimizing the total schedule time by implementing efficient scheduling techniques. The register allocation became then constrained by the total schedule time. Both the problems of minimizing the register requirement with a fixed total schedule time, or minimizing the total schedule time with a fixed number of registers was proven NP-hard [EGS95][2], even for DAGs. However, in some special cases, we can solve this problem with a polynomial complexity. For instance, Meleis in [MD99] gave a polynomial algorithm which produced an optimal schedule of a binary tree given a fixed number of registers : the tree must not contain unary operations, the latency of the operations must be 1, and the machine is restricted to issue no more than

[1] No static ILP can be expressed by the generated code.

[2] In fact, these problems are NP-complete.

one memory operation and one arithmetic operation per time slot. He has also solved recently the same problem but with pipelined loads in [Mel01].

1.2 Motivation

This report address the problem of register pressure in cyclic data dependence graphs (DDGs), with multiple registers types and no assumption about operation latencies. Our aim is to "uncouple" the registers constraints and allocation from the scheduling process. The principal reason is that the register allocation process is more important, as an optimization issue, than code scheduling. This is because the code performance is far more sensitive to the memory access than to the fine-grain scheduling : a cache miss prevents the processor from achieving a high dynamic ILP, even if the scheduler has extracted it at compile time. Our approach is to take into account the registers constraints before the code scheduling without hurting the ILP or restricting the scheduler.

Our work addresses two different global schemes for handling the registers constraints :

1. the registers constraints are taken into account before code scheduling without carrying out any register allocation, see Figure 1.1.a. The purpose is to ensure that the scheduling process will not require more registers than the ones available in the target processor, and hence the subsequent register allocation would not introduce spill code. In order to achieve these goals, we study the two notions which define the registers pressure of a DDG :

   (a) the register saturation (RS) which is the maximum number of registers required for any software pipelined schedule of the DDG. If this factor is less than the number of available registers R (see Figure 1.2.a), then the register pressure has no effects on the scheduler, and the DDG is left as it is. However, if the saturation exceeds R (see Figure 1.2.b), we add serial arcs into the DDG to reduce it below the limit R while minimizing the increase of the critical cycle ;

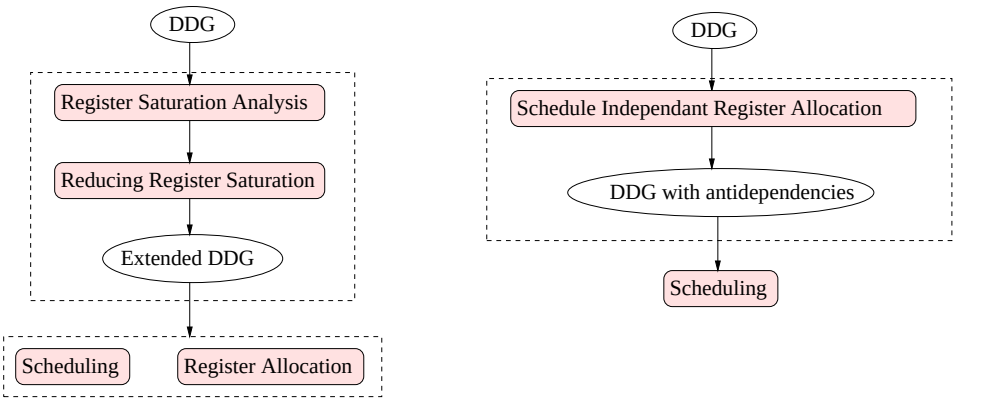
   (b) the register sufficiency (RF) which is the minimum number of registers required to pursue the computation of the loop. If this factor exceeds R (see Figure 1.2.c), then spill code cannot be avoided. We introduce these new memory access operations into the DDG in order to reduce RF below R.

   The principal motivation for this approach is that the registers constraints in modern processors are simpler than the resources constraints because there are few possible configurations : the number of registers can be 32, 64 or 128, and the register types that we generally focus on are classified as integer, float, guards, etc. However, resource constraints are less comparable. Each processor has its own properties. This obliges us to redo code scheduling for each target processor. With this approach, we can modify a DDG such that it satisfies the registers constraints for a whole set of processors with a number of available registers less or equal to register saturation. Our approach is portable since a lot of compilers (gcc for instance) uses the same first steps of compilation (parsing, intermediate code generation and optimization), but a different ILP backend for code optimization and generation for each target processor. This approach can be applied on

the intermediate code level to decide for register allocation without increasing the critical path if possible. After that, the ILP backend can schedule and optimize the code for a specific target processor.

2. the second approach consists of carrying out the cyclic register allocation before code scheduling under a fixed issue rate of a software pipelined schedule. This technique introduces some anti-dependencies between the operations because of the register reuse. Our purpose is that the introduced anti-dependencies do not prevent the scheduler from achieving a fixed initiation interval for the loop.

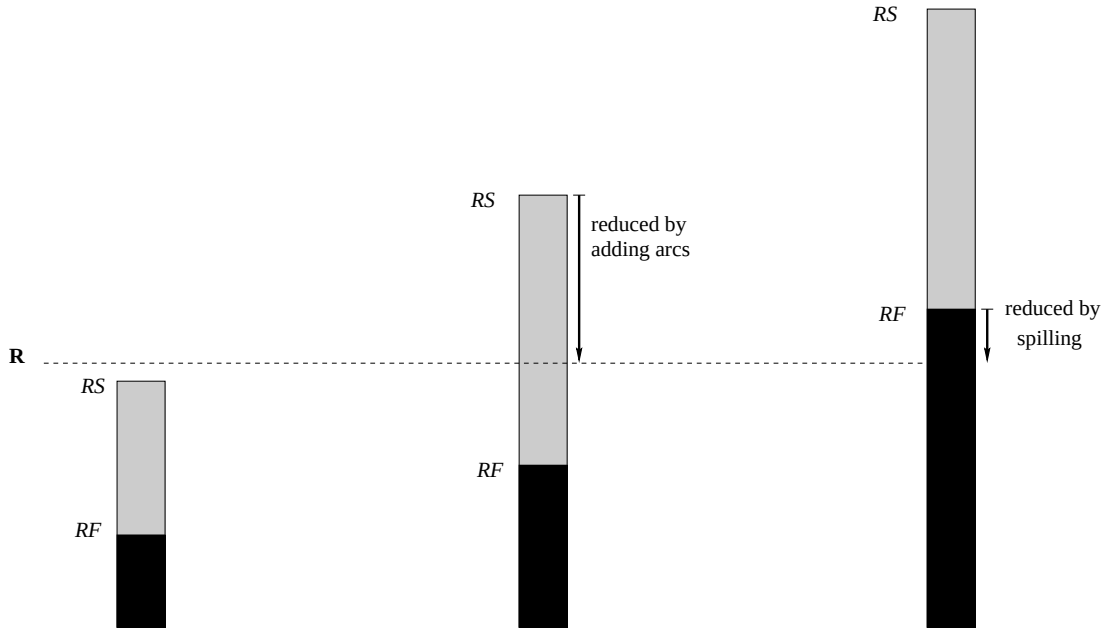
This report is organized as follows: Chapter 2 introduces our loop model and recalls the software pipelining optimization technique. In such periodic schedules, lifetime intervals may be circular. We also recall the cyclic register requirement and allocation of a SWP schedule. The cyclic register saturation is defined and studied in Chapter 3, where we give the exact formulation (using integer linear programming) and a heuristics for computing this quantity. Reducing the saturation by adding extra arcs is studied in this chapter too. Cyclic register sufficiency is defined and studied in Chapter 4. A heuristic for inserting spill code to reduce the sufficiency is described. Chapter 5 studies the cyclic register allocation before code scheduling under a fixed execution rate. Before concluding, we present the major work in the field of register allocation in Chapter 6.



(a) Register Saturation before Scheduling and Register Allocation

(b) Register Allocation before Scheduling

Figure 1.1: Steps of Register Pressure Analysis and Management



(a) Null Register Pressure (b) Possible Spill Code Insertion (c) Unavoidable Spill Code

Figure 1.2: Register Pressure Configurations

Chapter 2

Loop Model

This chapter introduces our DDG model. It consists of innermost loops without branches. As in the acyclic case, our model is sufficiently generic to be applied to both static and dynamic issue processors. We also recall software pipelining (SWP) method and how this strategy influences a late register allocation. The register need is slightly different in cyclic schedules since lifetime intervals become cyclic. We present an intLP formulation for it with $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E| + |V|^2)$ constraints, given a DDG $G = (V, E)$. The size of the constraints matrix is better than the complexity of the existing techniques which include an initiation interval factor.

This chapter is organized as follows. Section 2.1 defines our loop model (without branches) and presents our notations. Software pipelining technique is described in Section 2.2. We see that such periodic scheduling technique makes circular the values lifetimes intervals. Thus, the register requirement, studied in Section 2.3, is defined in a cyclic pattern. We present a method for computing it by decomposing the circular lifetime intervals into two classes : those which span the whole SWP kernel (correspond to different instances of the same statement), and those which span a fraction of the motif. We give an exact formulation of the cyclic register requirement according to an arbitrary SWP schedule using integer programming. This intLP system is used in further chapters for analyzing cyclic register saturation and sufficiency. Finally, before concluding with some remarks, Section 2.4 presents how a register allocation can be built cyclically on an already scheduled loop.

2.1 Definitions and Notations

A loop (without branches) in our study is represented by a graph $G = (V, E, \delta, \lambda)$ such that :

- V is the set of the statements in the loop body. Each statement u has a latency $lat(u) > 0$. The instance of the statement u (an operation) of the iteration i is noted $u(i)$. By default, the operation u denotes the operation $u(i)$;
- E is the set of precedence constraints (data dependences or other serial constraints) ;
- $\delta(e)$ is the latency of the arc e in terms of processor clock cycles ;
- $\lambda(e)$ is the distance of the arc e in terms of number of iterations. If $\lambda(e) > 0$, the dependence e is called *loop carried*. A valid schedule σ must satisfy :

$$\forall e = (u, v) \in E \ : \qquad \sigma\left(u(i)\right) + \delta(e) \leq \sigma\left(v(i + \lambda(e))\right)$$

We consider a target architecture with multiple registers types, where $\mathcal{T}$ denotes the set of register types (for instance, $\mathcal{T} = \{int, float\}$). We make a difference between statements and precedence constraints depending if they refer to values to be stored in registers or not :

1. $V_{R,t}$ is the set of values to be stored in registers of type $t \in \mathcal{T}$. We consider that each statement $u \in V$ writes into at most one register of a type $t \in \mathcal{T}$. The statements which define multiple values with different types are accepted in our model iff they do not define more than one value of a certain type. For instance, statements that produces one floating point result and one integer result are taken into account in our model. We denote by u^t the value of type t defined by the statement u;

2. $E_{R,t}$ is the set of flow dependence arcs through a value of type $t \in \mathcal{T}$. Since we accept the statements producing more than one value but with different types, these sets are not disjoined : for instance, we may have an arc $e \in E_{R,t_1} \cap E_{R,t_2}$.

To consider static issue processors (as VLIW) in which the hardware pipeline steps are made visible to compilers, we assume that reading from and writing into a register may be delayed from the beginning of the schedule time, and these delays are visible to the compiler (architectural visible). We define two delay (offset) functions $\delta_{r,t}$ and $\delta_{w,t}$ such that :

$$\begin{aligned} \delta_{w,t} : \quad & V_{R,t} \rightarrow \mathbb{N} \\ & u \mapsto \delta_{w,t}(u) / \ \delta_{w,t}(u) < lat(u) \\ & \text{the write cycle of } u^t \text{ into a register of type } t \text{ is } \sigma(u) + \delta_{w,t}(u) \end{aligned}$$

$$\begin{aligned} \delta_{r,t} : \quad & V \rightarrow \mathbb{N} \\ & u \mapsto \delta_{r,t}(u) / \ \delta_{r,t}(u) \leq \delta_{w,t}(u) < lat(u) \\ & \text{the read cycle of } u^t \text{ from a register of type } t \text{ is } \sigma(u) + \delta_{r,t}(u) \end{aligned}$$

Lastly, we assume that all the values produced in the loop are read at least once. A non consumed value in a loop is a statement which erases its result in the successive iterations, producing a self-output dependence with distance 1. If a non consumed value u^t exists in the loop, we can handle it in two ways :

1. we can assume that the statement u is removed from the loop by a previous dead code elimination process. Indeed, only the value produced at the last iteration has to be computed, and hence the operation $u(n)$ of the last iteration is inserted just after the loop;

2. since the value $u^t(i)$ is erased by $u(i+1)$, and hence killed by it, we can consider the self output dependence on u as a virtual self-*flow* dependence between u and itself with a distance 1 and a latency $\delta_{w,t}(u) + 1$ to model the fact that $u(i+1)$ consumes (kills) $u^t(i)$.

Till now, the best scheduling strategy of simple innermost loops is software pipelining (SWP). Next section gives a short description of SWP.

2.2 Software Pipelining

A software pipelined schedule σ of a graph $G = (V, E, \delta, \lambda)$ representing precedence constraints of a simple loop with n iterations, consists in overlapping the execution of the parallel operations belonging to different iterations [AJLA95]. A new iteration is initiated at constant rate

during the steady state before the (possible) completion of the previous one. The advantage of software pipelining is that optimal performance may be achieved with a more compact code size compared to loop unrolling followed by local scheduling. A SWP schedule is defined by an *initiation interval*[1] h and the schedule of the first iteration. Every h steps, a new iteration is issued. The schedule is written :

$$\forall u \in V,\ \forall i \in [1, n]\ :\qquad \sigma\left(u(i)\right) = \sigma_u + h \times i$$

where $\sigma\left(u(i)\right)$ is the schedule of the operation $u(i)$, and $\sigma_u = \sigma\left(u(1)\right)$ is the schedule of the operation u of the *first* iteration. The total schedule time of one iteration of the original loop body is then equal to $L = \max_{u \in V} \sigma_u$. Figure 2.1.(b) is an example of a software pipelined schedule with $h = 4$ of the DDG shown in part (a), in which the values and flow arcs are drawn with bold lines.

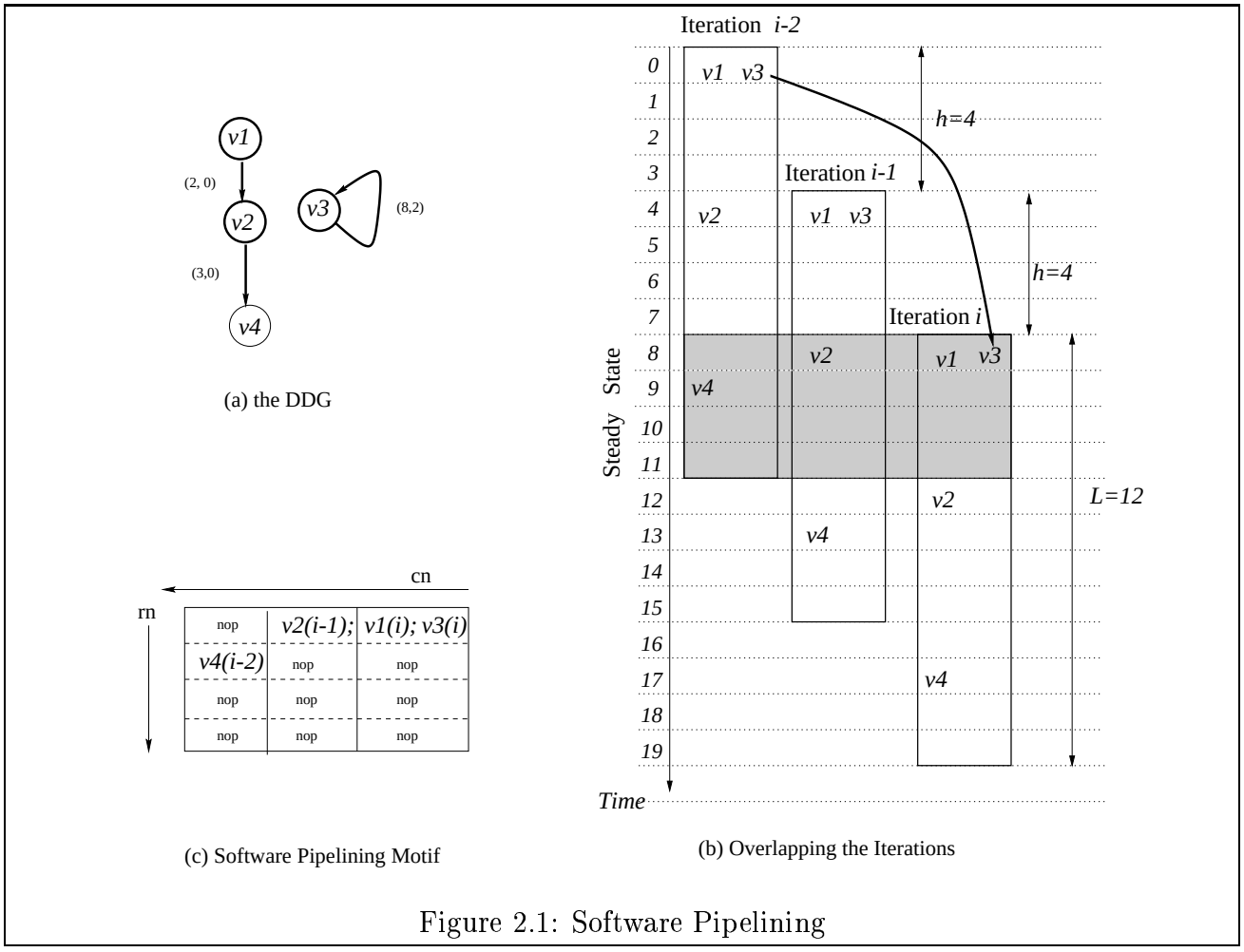


Figure 2.1: Software Pipelining

This periodic schedule defines a new compact loop body called the *motif* or the *kernel*. The successive iterations of the motif simulate the progression of the iterations of the original loop in a pipeline. Let $\Sigma(G)$ be the set of all valid software pipelined schedules of a loop G.

[1]Noted also II in some papers.

Also, we note $\Sigma_L(G)$ as the set of all valid software pipelined schedules with the property that the total schedule time of one iteration does not exceed L[2]:

$$\forall \sigma \in \Sigma_L(G),\ \forall u \in V, \qquad \sigma_u \leq L$$

For any $\sigma \in \Sigma(G)$, the minimum initiation interval MII, noted h_0 , is determined by the *critical circuit* of G, which defines the optimal execution rate. Let $\delta(C) = \sum_{e \in C} \delta(e)$ be the latency of the circuit C in G and $\lambda(C) = \sum_{e \in C} \lambda(e)$ its distance. Then a critical circuit C in G is defined by:

$$\frac{\delta(C)}{\lambda(C)} \geq \max_{C' \text{ a circuit in } G} \frac{\delta(C')}{\lambda(C')}$$

This critical ratio constitutes a lower limit for the minimal feasible initiation interval :

$$\forall \sigma \in \Sigma(G) : \qquad MII = \left\lceil \frac{\delta(C)}{\lambda(C)} \right\rceil \leq h$$

Note that the critical circuit can be computed with polynomial complexity algorithms ($\mathcal{O}(|V| \times |E| \times log\ |V|)$ [Law72, Saw97]). An implementation of an algorithm with the complexity $\mathcal{O}\big(|V|.|E|.log(|V|.\max_e \delta(e).\max_e \lambda(e)\big)$ is provided in [MN99].

If the critical ratio is not integral, this rate cannot be achieved. Nevertheless, e can avoid this loss of optimality by unrolling the loop j times before applying a periodic scheduling, where j is equal to the denominator of (rational) critical circuit cost to time ratio: the initiation interval of the unrolled loop becomes $j \times MII$.

If the DDG is acyclic, then $MII = 0$. This means that the loop is parallel (no circuit dependences): theoretically, we can completely unroll the loop an perform all iterations in parallel to obtain a maximal ILP[3]. We cannot assume such unbounded ILP degree scheduling because of code expansion and resource constraints. Since SWP focuses on building kernels, $MII = h_0$ is set to 1. Thus, the maximal number of parallel iterations is L.

The authors in [WEJS94] model the motif of a software pipelined schedule as a two dimensional matrix by defining a column number cn and row number rn for each statement. A SWP gets defined by three parameters, we note it $\sigma([rn], [cn], h)$. They define σ as:

$$\forall u \in V,\ \forall i \in [1, n] : \qquad \sigma\left(u(i)\right) = rn(u) + h \times \left(cn(u) + i\right)$$

where $cn(u) = \left\lfloor \frac{\sigma_u}{h} \right\rfloor$ and $rn(u) = \sigma_u\ mod\ h$.

Graphically, the row number $rn(u)$ is the step of the execution of the statement u relatively to the beginning of the motif, see Figure 2.1.(c): every h clock cycles, a new operation u is issued $rn(u)$ cycles after the beginning of the kernel. Statements that have the same row number are simply those that are issued in parallel. The column number $cn(u)$ represents the iteration number of the statement u, i.e. a statement u in the motif with a column number $cn(u)$ corresponds to the operation $u(i - cn(u))$ of the original loop. For example, a statement u with a null column number corresponds to the statement u of the original loop; a statement with a column number equal to 1 corresponds to the operation u of the iteration $i - 1$ of the

[2] L sufficiently large, i.e. greater than the critical path of the loop body.

[3] This maximal parallelism may be implemented at thread level, which is outside the scope of SWP for ILP.

original loop, etc.

Let us note B the acyclic data dependence graph of the loop body (G after removing the loop carried dependences). Then:

$$\forall \sigma \in \Sigma_L(G),\ \forall u \in V\ :\quad \underline{\sigma_u} \leq \sigma_u \leq \overline{\sigma_u}$$

in which:

- $\underline{\sigma_u} = LongestPathTo(u)$ is the as soon as possible schedule time of u in B;
- $\overline{\sigma_u} = L - LongestPathFrom(u)$ is the as late as possible schedule time of u in B according to the worst fixed total schedule time L of one original iteration.

We conclude that

$$\underline{cn}(u) \leq cn(u) \leq \overline{cn}(u)$$

where

$$\underline{cn}(u) = \left\lfloor \frac{\underline{\sigma_u}}{h} \right\rfloor ,\quad \overline{cn}(u) = \left\lceil \frac{\overline{\sigma_u}}{h} \right\rceil$$

In order to reach a steady execution state for the software pipelined loop, we need to fill the pipeline during a starting transient state. This is done by generating a *prologue* code before the SWP kernel. This prologue state lasts $L - h$ clock cycles so as to reach a maximal execution throughput for the pipelined execution of the loop (iterative execution of the kernel). Also, the last $L - h$ clock cycles of the total execution time correspond to an ending transient state in order to empty the pipeline: an *epilogue* code has to be generated, after the SWP motif, for this final state.

A value $u^t \in V_{R,t}$ is defined at the relative definition date $\sigma_u + \delta_{w,t}(u)$ clock cycles after the beginning of the motif. The killers of this value $u^t \in V_{R,t}$ are all the last scheduled consumers (readers). We note by $Cons(u^t)$ the set of the consumers of the value u^t. The last step when a value issued in the current motif is consumed is called the relative killing date:

$$k_\sigma(u^t) = \max_{\substack{v \in Cons(u^t) \\ e=(u,v) \in E_{R,t}}} \left(\sigma_v + \delta_{r,t}(v) + \lambda(e) \times h\right)$$

That is, the value $u^t(i)$ of the i^{th} iteration is defined at the absolute time $\sigma_u + \delta_{w,t}(u) + i \times h$ and killed at the absolute time $k_\sigma(u) + i \times h$.

In our model, we assume that a value written at instant c is alive one step later[4]. The relative acyclic *life interval* (range) of the value $u^t \in V_{R,t}$ is:

$$LT_\sigma(u^t) =]\sigma_u + \delta_{w,t}(u), k_\sigma(u^t)]$$

The absolute life interval of the value $u^t(i)$ is:

$$]\sigma_u + \delta_{w,t}(u) + i \times h, k_\sigma(u^t) + i \times h]$$

The *lifetime* of a value $u^t \in V_{R,t}$ is the total number of clock cycles during which this value is alive according to the schedule:

$$lifetime_\sigma(u^t) = k_\sigma(u) - \sigma_u - \delta_{w,t}(u)$$

[4]This is not a limitation on the model, but a choice.

2.3 Cyclic Register Need

The cyclic register need (also known in the literature as register requirement or MAXLIVE) of type t is the maximum number of values of that type which are simultaneously alive in the software pipelining motif. In the case of a cyclic schedule, some values may be alive during many iterations and different copies (instances) of the values may interfere. Figure 2.2 illustrates another schedule of the DDG previously shown in Figure 2.1.(a) : the value v_3 interferes with itself[5].

Lifetime intervals during the steady state describe a circular lifetime interval graph around the motif: we "wrap" a circle of circumference h by the acyclic lifetime intervals of values. Then, the lifetime intervals are cyclic.

Definition 2.1 (Circular Lifetime Interval) *A circular lifetime interval produced by wrapping a circle of circumference h by an interval $I =]a, b]$ is defined by a triplet of integers (l, r, p), such that:*

- *$l = a \bmod h$ is called the* **l***eft of the cyclic interval;*
- *$r = b \bmod h$ is called the* **r***ight of the cyclic interval;*
- *$p = \lfloor \frac{b-a}{h} \rfloor$ is the number of complete* **p***eriods (turns) around the circle, which corresponds to the number of interfering copies.*

As an example, the circular lifetime interval of v_1 in Figure 2.2.(b) is $(1, 3, 0)$, v_2's one is $(2, 1, 0)$ and v_3's one is $(2, 0, 1)$.

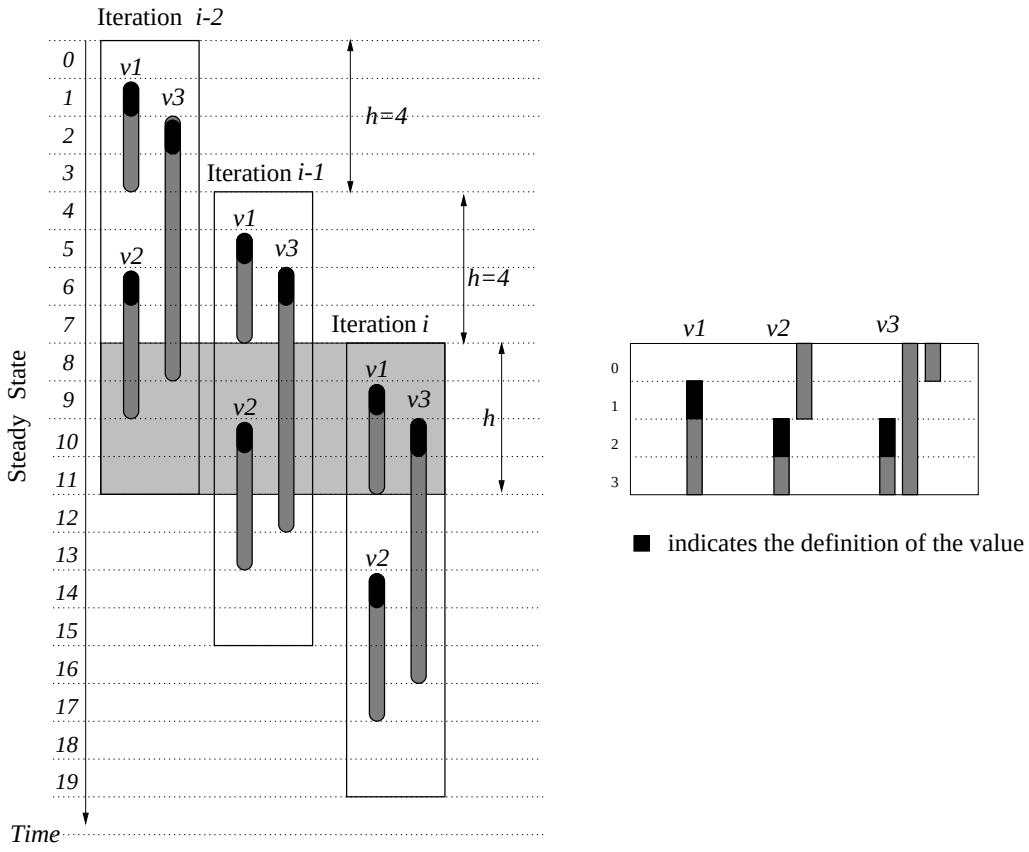
The set of all the circular lifetime intervals around the motif defines a circular interval graph which we note $C_h(G)$. In this thesis, we use the short term of circular interval to indicate a circular lifetime interval, and the term of circular graph for indicating a circular lifetime intervals graph. Figure 2.3.(a) gives an example of a circular graph. The maximum number of values simultaneously alive is the width of this circular graph, i.e. the maximum number of circular intervals which interfere at a certain point of the circle. For instance, the width of the circular graph of Figure 2.3.(a) is 4. Figure 2.2.(b) is another representation of the circular graph when we cut the circle at the instant 0.

Definition 2.2 (Cyclic Register Need (Requirement)) *Let $G = (V, E, \delta, \lambda)$ be a loop and $\sigma \in \Sigma(G)$ a software pipelined schedule. The cyclic register need of type $t \in \mathcal{T}$ is the width of the circular graph produced by wrapping the lifetime intervals of type t around a period h. We note it $CRN_t^\sigma(G)$.*

We call *circular excessive values* a set of a maximum number of values simultaneously alive. In Figure 2.2.(b) for instance, $v_1(i), v_3(i), v_3(i-1)$ and $v_2(i-1)$ are circular excessive values.

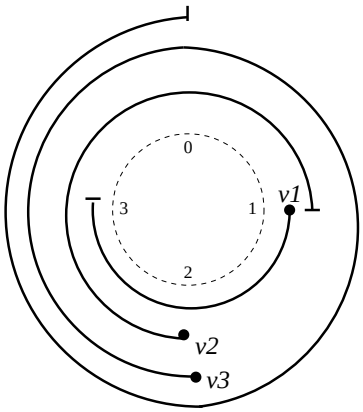
Computing the width of a circular interval graph is obvious. We can compute the number of values simultaneously alive at each clock cycle in the SWP kernel. This leads to a method of which complexity depends on the initiation interval h. This factor may be very large since it depends on the specified latencies in the DDG, and on its structure (critical circuit). We want

[5]Remember that the lifetime intervals are left open and right closed.

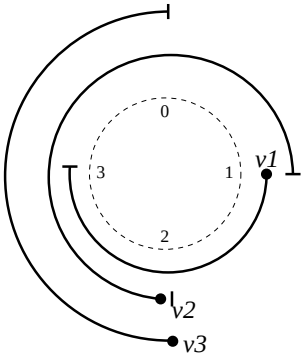


(a) Software Pipelining (b) Life Intervals inside the Motif

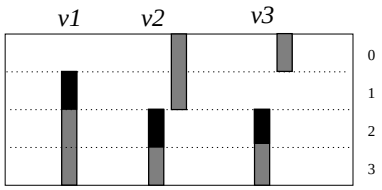
Figure 2.2: Cyclic Register Need in Software Pipelining Schedules



(a) Circular Life Intervals Graph



(b) in_fraction_of_h Circular Graph



(c) in_fraction_of_h intervals

Figure 2.3: Circular Life Intervals Graph

to provide a better method of which complexity only depends on DDG size, i.e. only depends on the number of statements and dependencies. For this purpose, we study the relationship between the width of a circular interval graph with the size a maximal clique in the interference graph[6]. The following paragraphs are devoted to this aim.

In general, the width of a circular interval graph is not equal to the size of a maximal clique in the interference graph [Tuc75]. In order to effectively compute this width, we decompose the circular graph $C_h(G)$ into two parts.

1. The first part is the number of the complete turns around the circle, which corresponds to the number of copies of each value that are simultaneously alive at all times during the steady-state portion of the cyclic schedule : $\sum_{(l,r,p) \text{ a circular interval}} p$.
2. The second part is composed of the remainder of the lifetime intervals after removing all the complete turns (see Figure 2.3.(b) and (c)). The size of the remaining intervals is strictly less than h, the size of the SWP motif. Note that if the left of a circular interval is equal to its right ($l = r$), then the remaining interval after ignoring the complete turns around the circle is empty ($]l, r] =]l, l] = \phi$). These empty intervals are removed from this second part. Two classes of intervals remain.
   (a) The first class contains acyclic intervals that do not cross the kernel barrier, i.e. when the left is less than the right ($l < r$). v_1 in Figure 2.3.(b) and (c), for instance, belongs to this class.
   (b) The second class contains intervals that cross kernel barrier, i.e. when the left is greater than the right ($l > r$). v_2 and v_3 in Figure 2.3.(b) and (c), for instance, belong to this class. These acyclic intervals represent the left and the right parts of the lifetime intervals. When merging the left and right parts of a value of two successive SWP motifs, we create a new contiguous circular interval.

   These two classes of intervals define a new circular graph. We call it an *in_fraction_of_h* [Alt95] circular graph because the size of its lifetime intervals is less than h. This circular graph contains the circular intervals of the first class, and those of the second class after merging the left of each value with its right.

Definition 2.3 (in_fraction_of_h Circular Graph) *Let $C_h(G)$ be a circular graph of a loop $G = (V, E, \delta, \lambda)$. The in_fraction_of_h lifetime interval graph noted $\underline{\overline{C_h}}(G)$ is the circular graph after ignoring the complete turns around the circle :*

$$\underline{\overline{C_h}}(G) = \{(l, r, 0) \ / \ \exists p, (l, r, p) \in C_h(G) \ \wedge \ r \neq l\}$$

We call the circular interval $(l, r, 0)$ a *circular in_fraction_of_h interval*, and we can simply note it (l, r). Any circular interval in $(l, r) \in \underline{\overline{C_h}}(G)$ has a length lower than h clock cycles. Then, the total cyclic register need becomes :

$$CRN_t^{\sigma}(G) = \sum_{(l,r,p) \in C_h(G)} \left[p + w\left(\underline{\overline{C_h}}(G)\right) \right]$$

[6]Remember that the interference graph is an undirected graph that models interference relations between lifetime intervals : two statements u and v are connected iff their (circular) lifetime intervals share a unit of time.

where w denotes the width of the in_fraction_of_h circular graph.

As stated before, in a general circular graph, the size of a maximal clique in the interference graph is not equal to its width. To overcome this problem, we use the fact that the in_fraction_of_h circular graph $\overline{C_h}(G)$ has circular intervals which do not make complete turns around the circle. Then, if we unroll the motif twice to consider the values produced during two successive periods of the kernel, the complete interference pattern is exhibited. For instance, the circular graph of Figure 2.4.(a) has a width equal to 2. Its interference graph in Figure 2.4.(b) has a maximal clique of size 3. Since the size of these intervals does not exceed a period h, we unroll twice the circular graph like shown in Figure 2.4.(c). The interference graph of the acyclic intervals in Figure 2.4.(d) has a size of a maximal clique equal to the width 2. The following theorem proves this fact.

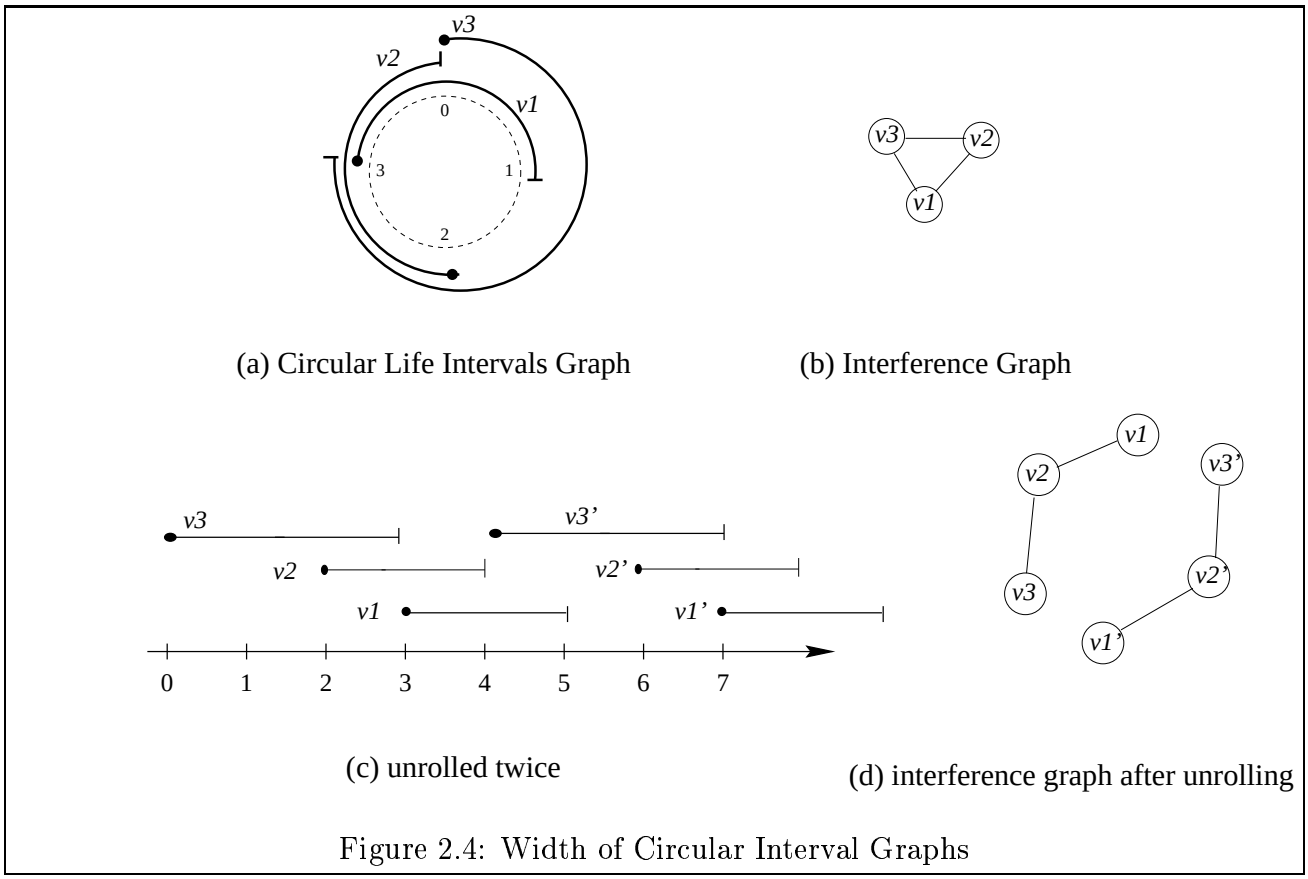
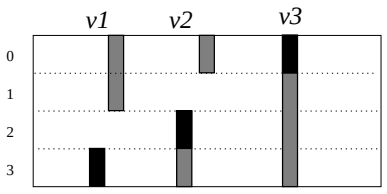


Figure 2.4: Width of Circular Interval Graphs

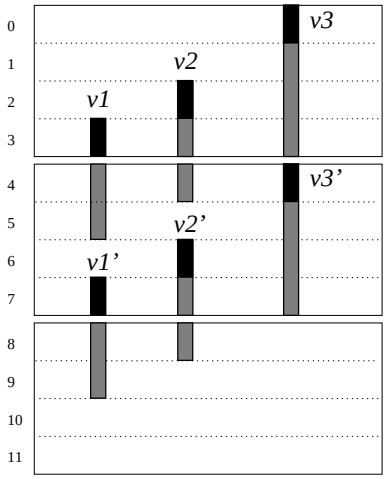
Theorem 2.1 *Let $\overline{C_h}(G)$ be a circular in_fraction_of_h graph (no complete turns around the circle exist). For each circular in_fraction_of_h interval $(l, r) \in C_h(G)$, we create the two corresponding acyclic intervals I and I' after merging the lefts and the rights of two successive kernels. Then, the cardinality of any maximal clique in the interference graph of all these acyclic intervals is equal to the width of $\overline{C_h}(G)$.*

Proof :

We will prove that the width of the acyclic in_fraction_of_h intervals after unrolling the kernel twice is equal to the width of $\overline{C_h}(G)$, i.e. the number of values simultaneously alive in the kernel. Figure 2.5 gives an example.



(a) in_fraction_of_h intervals



(b) acyclic in_fraction_of_h intervals

Figure 2.5: Acyclic in_fraction_of_h intervals

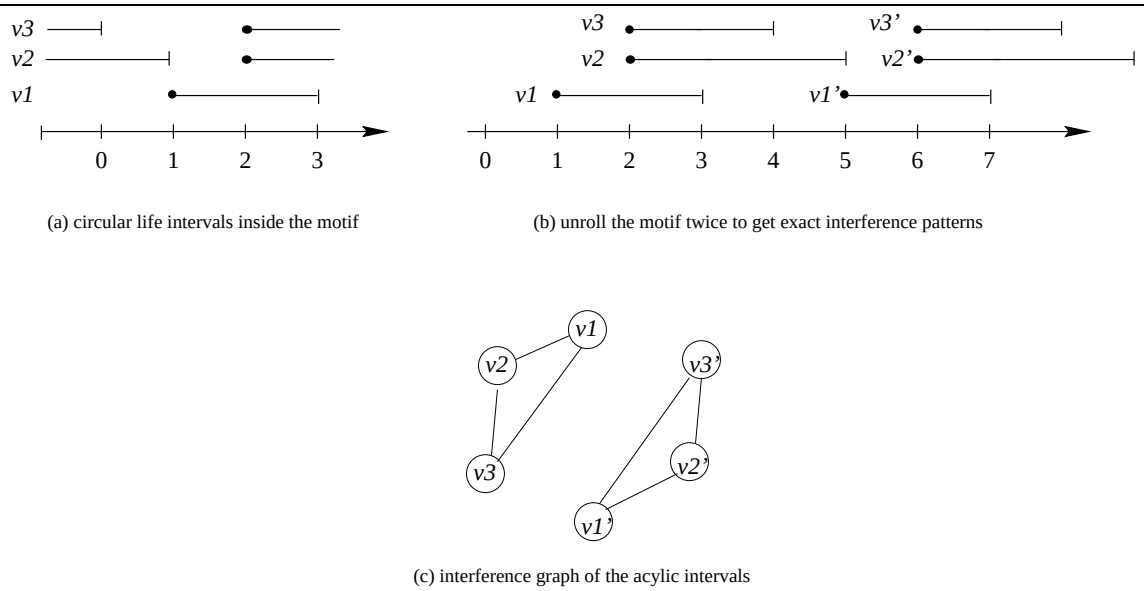
The acyclic in_fraction_of_h intervals belong to a whole window $[0, 3h[$. Note that for each circular interval $(l, r) \in \overline{C_h}(G)$, its corresponding acyclic intervals I and I' cannot interfere because their length is less than h . $I \prec I'$ because :

- if $r \geq l$ i.e. $I =]l, r]$ and $I' =]l + h, r + h]$, then $(l + h) - r \geq 0$ because the length $r - l < h$
- if $r < l$ i.e. $I =]l, r + h]$ and $I' =]l + h, r + 2 \times h]$, then the length $r + h - l < h$ and hence $(l + h) - (r + h) \geq 0$;

Now, let examine the whole window $[0, 3h[$, as shown in Figure 2.5.(b) :

1. the window $[0, h[$ contains the intervals of the original kernel except the rights which go throught the h barrier (for instance, the rights of v_1 and v_2). So the width of the intervals during this window is less or equal than the width of the circular intervals in the original kernel ;
2. the window $[h, 2h[$ contains exactly the intervals of the original kernel. So the width here is the same as the original ;
3. the window $[2h, 3h[$ contains the intervals of the original kernel except the lefts of the intervals coming from the previous window $[h, 2h[$ (for instance, the lefts of v'_1 and v'_2). So the width of the intervals during this window is less or equal than the original ;

We conclude that during the window $[0, 3h[$, the number of values simultaneously alive is exactly the same as in the original kernel. Hence, any maximal clique in the interference graph of the acyclic in_fraction_of_h intervals corresponds to a maximal set of values simultaneously alive in the circular in_fraction_of_h graph.



(a) circular life intervals inside the motif

(b) unroll the motif twice to get exact interference patterns

(c) interference graph of the acylic intervals

Figure 2.6: The Width is the Maximal Clique after Unrolling Twice

We call such an acyclic interval an *acyclic in_fraction_of_h interval*. Given a circular in_fraction_of_h interval $(l, r) \in C_h(G)$, the two corresponding acyclic in_fraction_of_h intervals are :

- $I =]l, r]$ and $I' =]l + h, r + h]$ if $r \geq l$;
- $I =]l, r + h]$ and $I' =]l + h, r + 2 \times h]$ if $r < l$;

Figure 2.6 shows the unrolled circular graph of the in_fraction_of_h circular graph previously described in Figure 2.3 on page 12. The interference graph is an interval graph, and hence the maximal clique can be computed with a $\mathcal{O}(n \times log(n))$ complexity [Ber77]. So, we have defined a method that computes the cyclic register need in which complexity depends only on the size of the input DDG. The complete turns around the circles is computed in linear time ($\mathcal{O}(|V|)$), and the width of the in_fraction_of_h graph is computed with a complexity $\mathcal{O}(|V| \times log(|V|))$.

Note that if the length of a circular interval (l, r, p) is a multiple of h, then its in_fraction_of_h interval is empty since $l = r$ (lifetime intervals are open from the left). Consequently, it is removed from the set of in_fraction_of_h intervals. As an illustration, the circular interval of v_3 in Figure 2.7 has $lifetime(v_3) = 4$ with $h = 4$. Its corresponding in_fraction_of_h interval is $(0, 0)$. This latter corresponds to two empty acyclic in_fraction_of_h intervals $]0, 0]$ and $]4, 4]$. They must be removed from the set of acyclic in_fraction_of_h intervals.

When looking for a software pipelined schedule with a limited register need, choosing a "suitable" initiation interval is a crucial issue. It is intuitive that lower the initiation interval h is, higher the register pressure is, since more parallelism requires more memory. If we succeed in finding a software pipelined schedule which needs R registers, then it is possible to get another

software pipelined schedule which needs R registers with a higher II if we relax the uper-bound L.

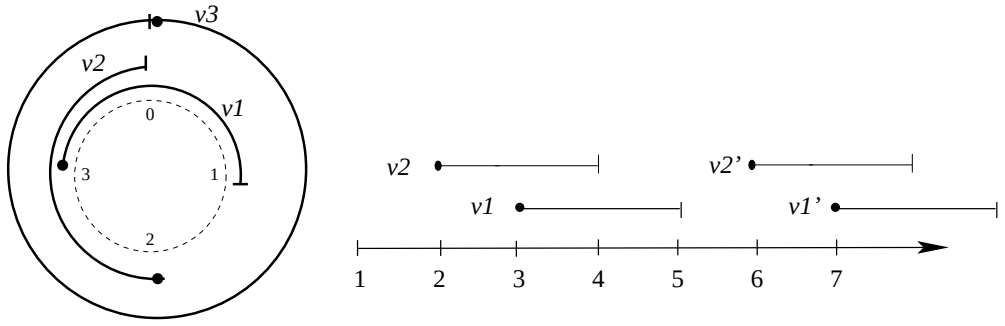
Proposition 2.1 *Let $G = (V, E, \delta, \lambda)$ be a DDG of a loop. If there exists a software pipelining $\sigma([rn], [cn], h)$ which needs R registers of type t with $h \leq L$, then there exists a software pipelining $\sigma'([rn'], [cn'], h+1)$ which needs R registers of type t with $L' = L + 1 + \lfloor L/h \rfloor$. Formally:*

$$\forall \sigma([rn], [cn], h) \in \Sigma_L(G)/h_0 \leq h \leq L,$$
$$\exists \sigma'([rn'], [cn'], h+1) \in \Sigma_{L+1+\lfloor L/h \rfloor}(G) : CRN_t^{\sigma'}(G) = CRN_t^{\sigma}(G)$$

Proof:

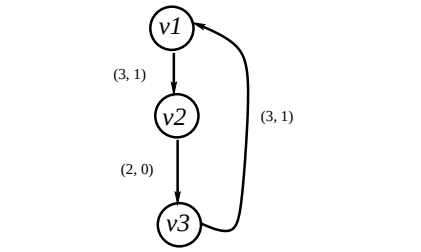
Let be $\sigma([rn], [cn], h)$ a valid software pipelined schedule for G. In this proof, we show how to construct another schedule $\sigma'([rn'], [cn'], h+1)$ which needs the same number of registers as σ. For the simplicity of this proof and without loss of generality, let neglect the reading and writing delays, i.e. we set $\delta_r = \delta_w = 0$. We use the example in Figure 2.8 to illustrate this proof: Figure 2.8.(c) is the kernel of a valid σ for the loop of Figure 2.8.(a) with null writing and reading delays, where the execution rate is $h = 4$. Figure 2.8.(b) shows the pipelined execution of the loop. The cyclic register need is 2 as shown by the lifetime intervals in the motif, Figure 2.8.(d). To construct another σ' with $h' = 5$, we proceed by inserting a complete row of static nops in the kernel of σ without changing the maximal number of values simultaneously alive, while preserving the validity of the schedule.

To add the row of nops in the kernel of σ, we choose an excessive clock cycle $0 \leq c < h$, i.e. a clock cycle with a maximal number of values simultaneously alive, as the clock cycle 1 in Figure 2.8.(d). Note that we can have more than one excessive clock cycle, any of them is a suitable candidate. Then, we "shift" down by one clock cycle all the statements scheduled by σ during or after the row c, as illustrated in Figure 2.8.(e). We let the other statements unchanged (those scheduled strictly before the row c). The new kernel has the following properties:

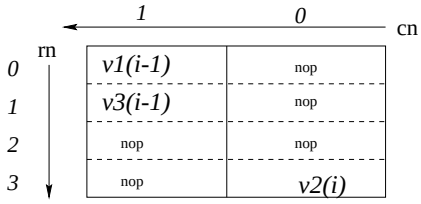


(a) Circular Life Intervals Graph (b) In_fraction_of_h intervals

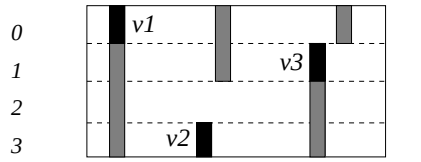
Figure 2.7: Empty in_fraction_of_h Intervals



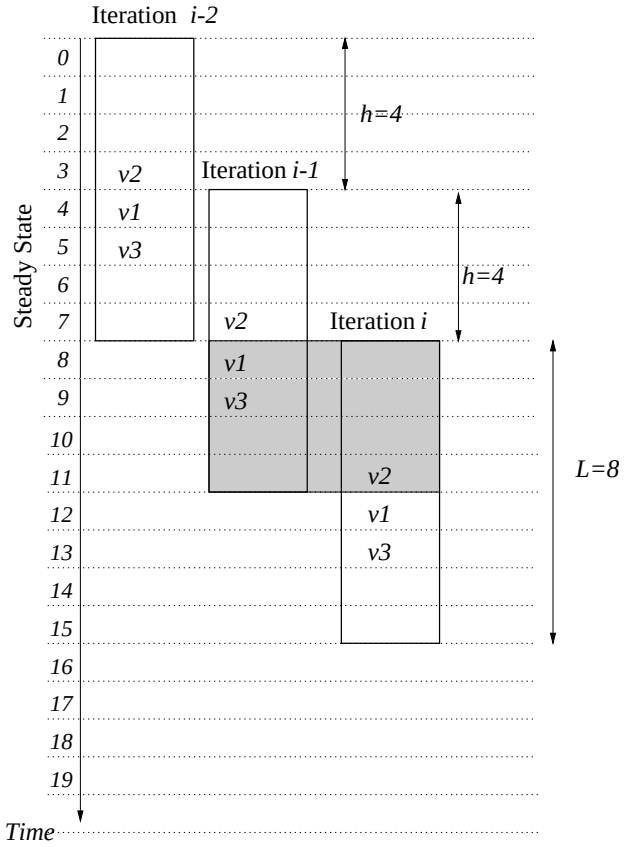
(a) the DDG



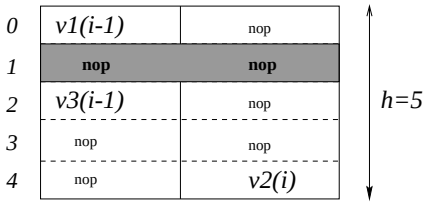
(c) Software Pipelining Motif



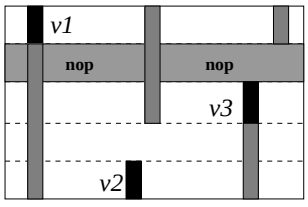
(d) Lifetime intervals



(b) Overlapping the Iterations



(e) Adding one row of nops ...



(f) ... does not increment the register need

Figure 2.8: Adding a row of nops does not change the register requirement

1. its initiation interval is $h' = h + 1$ since we have shifted down the considered statements by only one clock cycle;
2. the values simultaneously alive during the row c in the previous kernel are exactly the same in the new kernel:
3. the lifetime intervals of the values simultaneously alive during c in the previous kernel have been lengthened by 1 in the new kernel;
4. the interferences between the lifetime intervals remain unchanged in the new kernel, see Figure 2.8.(f): our transformation is similar to considering that the clock cycle $c-1$ has been decomposed into two virtual clock cycles.

Now, we have to prove that the constructed schedule is valid. Formally, the new schedule $\sigma'([rn'], [cn'], h+1)$ is defined by $[cn'] = [cn]$ and: $\forall u \in V$

$$rn'(u) = \begin{cases} rn(u) & \text{if } rn(u) < c \\ rn(u) + 1 & \text{otherwise} \end{cases}$$

σ is a valid schedule means that (see [Saw97]) $\forall e = (u, v)$:

$$rn(v) - rn(u) + h(\lambda(e) + cn(v) - cn(u)) \geq \delta(e)$$

and $\lambda(e) + cn(v) - cn(u)) \geq 0$. We have to verify that: $\forall e = (u, v) \in E$

$$rn'(v) - rn'(u) + (h+1)(\lambda(e) + cn'(v) - cn'(u)) \geq \delta(e) \tag{2.1}$$

There are four cases:

1. if $rn(u) < c$ and $rn(v) < c$ then $rn(u)' = rn(u)$ and $rn'(v) = rn(v)$. Since $[cn'] = [cn]$, Equation 2.1 becomes:

   $$rn(v) - rn(u) + h(\lambda(e) + cn(v) - cn(u)) + (\lambda(e) + cn(v) - cn(u)) \geq \delta(e)$$

   verified because $\lambda(e) + cn(v) - cn(u)) \geq 0$ and σ is valid;
2. if $rn(u) < c$ and $rn(v) \geq c$ then $rn(u)' = rn(u)$ and $rn'(v) = rn(v) + 1$. Equation 2.1 becomes:

   $$rn(v) - rn(u) + h(\lambda(e) + cn(v) - cn(u)) + (\lambda(e) + cn(v) - cn(u)) + 1 \geq \delta(e)$$

   verified because $\lambda(e) + cn(v) - cn(u)) \geq 0$ and σ is valid;
3. if $rn(u) \geq c$ and $rn(v) < c$ then $rn(u)' = rn(u) + 1$ and $rn'(v) = rn(v)$. The fact that $rn(u) > rn(v)$ means that the dependence $e = (u, v)$ is necessarily an inter kernel one (inter iteration), i.e. verified by the successive iterations of the motif. Then, $\lambda(e) + cn(v) - cn(u)) > 0$ necessarily [Saw97]. Equation 2.1 becomes:

   $$rn(v) - rn(u) + h(\lambda(e) + cn(v) - cn(u)) + (\lambda(e) + cn(v) - cn(u)) - 1 \geq \delta(e)$$

   verified because $\lambda(e) + cn(v) - cn(u)) > 0$ and σ is valid;

4. if $rn(u) \geq c$ and $rn(v) \geq c$ then $rn(u)' = rn(u)$ and $rn'(v) = rn(v)$. Equation 2.1 becomes:

   $$rn(v) - rn(u) + h(\lambda(e) + cn(v) - cn(u)) + (\lambda(e) + cn(v) - cn(u)) + 1 - 1 \geq \delta(e)$$

   verified because $\lambda(e) + cn(v) - cn(u)) \geq 0$ and σ is valid;

Finally, let us compute the upper-bound L' of one iteration with the constructed schedule σ'. $\forall u \in V$:

$$\begin{aligned} \sigma'_u &= rn'(u) + cn'(u) \times h' \\ &= rn'(u) + cn(u) \times (h+1) \\ &\leq rn(u) + cn(u) \times h + 1 + cn(u) \\ &\leq \sigma_u + 1 + \lfloor L/h \rfloor \\ &\leq L + 1 + \lfloor L/h \rfloor = L' \end{aligned}$$

⌟

Computing the cyclic register need of a SWP is easy: we build the circular lifetime graph and we compute its width. However, we need to formulate it according to an arbitrary SWP, i.e. without fixing any scheduling information. Next section gives an exact intLP formulation of $CRN_t(G)$ according to a variable schedule. This formulation enables us in further chapters to compute exact register pressure.

2.3.1 Exact Formulation of Cyclic Register Need

A "good" exact intLP model is important in our study because it must be used further for maximizing (saturation) or minimizing (sufficiency) the cyclic register need, and if possible, with the same variables and constraints. Furthermore, we need to give a "good" intLP complexity in terms of the number of generated variables and constraints. This complexity must be a polynomial function of the size of input DDG, i.e. it must only depend on the number of nodes and arcs without introducing the II factor like in existing techniques.

In this section, we show how to model the exact register requirement of arbitrary cyclic schedules. For this purpose, we use Theorem 2.1 which consists in unrolling twice the kernel to exhibit the complete interference pattern between the in_fraction_of_h intervals. In our exact model, we suppose the following constants:

- L: a worst total schedule time of one iteration;
- h: the initiation interval.

Basic Variables

1. For lifetime intervals, we define:
   - one schedule variable σ_u for each $u \in V$;
   - one variable which contains the killing date k_{u^t} for each $u^t \in V_{R,t}$.
2. For cyclic register need, we define:

- p_{u^t} the number of the copies of $u^t \in V_{R,t}$ simultaneously alive, which is the number of the complete turns around the circle produced by $u^t \in V_{R,t}$;
- l_{u^t} and r_{u^t} the left and the right of the cyclic lifetime interval of $u^t \in V_{R,t}$;
- the two acyclic in_fraction_of_h intervals $I_{u^t} =]a_{u^t}, b_{u^t}]$ and $I'_{u^t} =]a'_{u^t}, b'_{u^t}]$ after unrolling the kernel twice.

3. For a maximal clique in the interference graph of the in_fraction_of_h acyclic intervals, we define :
   - interference binary variables $s_{I,J}$ for all the in_fraction_of_h acyclic intervals I, J : $s_{I,J} = 1$ iff I and J interfere with each other ;
   - a binary variable x_I for each in_fraction_of_h acyclic interval : $x_I = 1$ iff I belongs to a maximal clique.

Linear Objective Function

The cyclic register requirement of type t is the maximal of :

$$\sum_{\text{acyclic in\_fraction\_of\_}h\text{ interval } I} x_I + \sum_{u^t \in V_{R,t}} p_{u^t}$$

As we will see soon, maximizing this function amounts to computes the cyclic register saturation (CRS). It can also be used to compute cyclic register sufficiency (CRF).

Linear Constraints

1. Cyclic scheduling constraints :

$$\forall e = (u, v) \in E \ : \qquad \sigma_u + \delta(e) \leq \sigma_v + \lambda(e) \times h$$

2. The killing dates are computed by :

$$\forall u^t \in V_{R,t} \ : \qquad k_{u^t} = \max_{\substack{v \in Cons(u^t) \\ e=(u,v) \in E_{R,t}}} \left(\sigma_v + \delta_{r,t}(v) + \lambda(e) \times h\right)$$

   We use the linear constraints of the "maximum" defined in [Tou01d, Tou01a, Tou01c]. k_{u^t} is bounded by $\underline{k_{u^t}}$ and $\overline{k_{u^t}}$ where :
   - $\underline{k_{u^t}} = \min_{v \in Cons(u^t)} \left(\underline{\sigma_v} + \delta_{r,t}(v) + \max_{e=(u,v) \in E_{R,t}} \lambda(e) \times h\right)$
   - $\overline{k_{u^t}} = \max_{v \in Cons(u^t)} \left(\overline{\sigma_v} + \delta_{r,t}(v) + \max_{e=(u,v) \in E_{R,t}} \lambda(e) \times h\right)$

3. The number of copies of a value (complete turns around the circle) is the integer division of the lifetime by h. We introduce an integer variable α_{u^t} which holds the rest of the division :

$$\begin{cases} k_{u^t} - \sigma_u - \delta_{w,t}(u) = p_{u^t} \times h + \alpha_{u^t} \\ \alpha_{u^t} < h \\ \alpha_{u^t} \in \mathbb{N} \end{cases}$$

4. The lefts of the circular intervals are the rest of the integer division of the definition date by h. We introduce an integer variable β_{u^t} which holds the integer quotient of the division:

$$\begin{cases} \sigma_u + \delta_{w,t}(u) = \beta_{u^t} \times h + l_{u^t} \\ l_{u^t} < h \\ \beta_{u^t} \in \mathbb{N} \end{cases}$$

5. The rights of the circular intervals are the rest of the integer division of the killing date by h. We introduce an integer variable γ_{u^t} which holds the integer quotient of the division:

$$\begin{cases} k_{u^t} = \gamma_{u^t} \times h + r_{u^t} \\ r_{u^t} < h \\ \gamma_{u^t} \in \mathbb{N} \end{cases}$$

6. The in_fraction_of_h acyclic intervals are computed by unrolling the kernel twice, depending if the cyclic interval crosses the kernel barrier (Theorem 2.1):

$$\begin{cases} a_{u^t} = l_{u^t} \\ r_{u^t} \geq l_{u^t} \Longrightarrow b_{u^t} = r_{u^t} \\ r_{u^t} < l_{u^t} \Longrightarrow b_{u^t} = r_{u^t} + h \quad \text{(case when the cyclic interval crosses } h\text{)} \\ a'_{u^t} = a_{u^t} + h \\ b'_{u^t} = b_{u^t} + h \end{cases}$$

We use the linear constraints of implication defined in [Tou01d, Tou01a, Tou01c] since the variable domains are bounded. We know that $0 \leq l_{u^t} < h$, so $0 \leq a_{u^t} < h$ and $h \leq a'_{u^t} < 2h$. Also, $0 \leq l_{u^t} < h$ so $0 \leq b_{u^t} < 2h$ and $h \leq b'_{u^t} < 3h$.

7. The interference binary variables $s_{I,J}$ are computed as in the acyclic case ([Tou01d, Tou01a, Tou01c]), except that we must check if acyclic in_fraction_of_h intervals are not empty. We have to express in the intLP the following constraints.
$\forall$ acyclic intervals I, J:

$$s_{I,J} = 1 \Longleftrightarrow \left[(length(I) > 0) \ \wedge \ (length(J) > 0) \ \wedge \ \neg(I \prec J \ \vee \ J \prec I)\right]$$

where $\prec$ denotes the relation *before* in the interval algebra. Assuming that $I =]a_I, b_I]$ and $J =]a_J, b_J]$, these constraints are written as follows. $\forall$ acyclic intervals I, J:

$$s_{I,J} = 1 \Longleftrightarrow \begin{cases} b_I - a_I > 0 & (i.e.\ length(I) > 0) \\ b_J - a_J > 0 & (i.e.\ length(J) > 0) \\ b_I > a_J & (i.e.\ \neg(I \prec J)) \\ b_J > a_I & (i.e.\ \neg(J \prec I)) \end{cases}$$

8. A maximal clique in the interference graph is an independent set in the complementary graph. Then, for two binary variables x_I and x_J, only one is set to 1 if the two acyclic intervals I and J do not interfere with each other:

$$\forall \text{ acyclic intervals } I, J: \qquad s_{I,J} = 0 \Longrightarrow x_I + x_J \leq 1$$

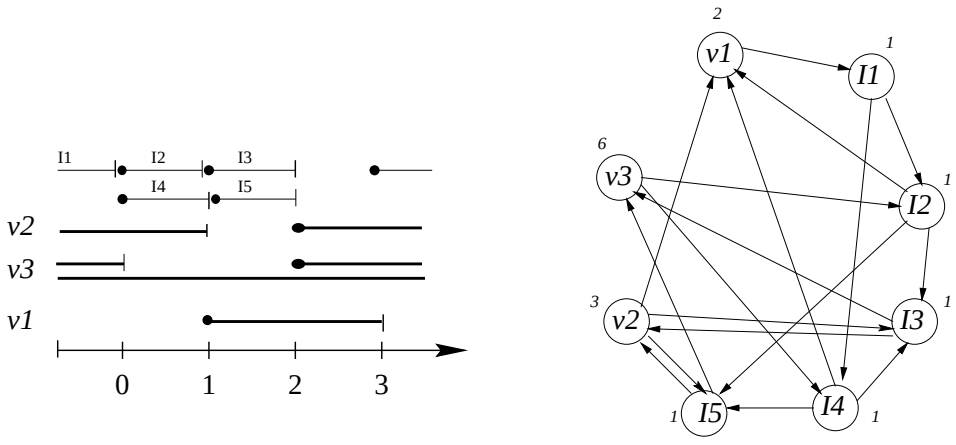
Note that we can optimize this intLP model since the two acyclic in_fraction_of_h intervals I_{u^t} and I'_{u^t} of a value u^t cannot interfere, and hence we do not define nor compute $s_{I,I'}$. Similar optimization can be done regarding redundant arcs or impossible interfering relations detected at compile time (statically).

Register allocation for acyclic scheduled codes is obvious because lifetime intervals are defined. However, the cyclic case is slightly different because values produced by the same statement may interfere. Next section presents register allocation of software pipelined loops.

2.4 Register Allocation of Software Pipelined Loops

This section gives a brief description of the meeting graph (MG) framework [ELM95, ELM97, dWELM99, Lel96] intended for cyclic register allocation of already scheduled loops. The meeting graph is based on a circular lifetime intervals graph. If w is the width of the circular graph, the problem is to allocate w available registers (colors) to the circular intervals. Without loss of generality, the width w of the circular intervals is assumed constant around the circle. If it is not really the case, it is always possible to add unit-time fictitious intervals around the circle where the width is less than w, as in Figure 2.9. This example is the MG of the circular intervals previously presented in Figure 2.2 and Figure 2.3.

Definition 2.4 (Meeting Graph) *Let $C_h(G)$ be a circular lifetime interval graph for a register type t with a constant width w. The meeting graph related to $C_h(G)$ of the register type t is a directed weighted graph $M_t = (V_{R,t}, E_M, \omega)$. There is an arc between u^t and v^t in M_t iff the circular lifetime interval of u^t ends when that of v^t begins. Each $u^t \in V_{R,t}$ is weighted by $\omega(u^t) = lifetime(u^t)$.*



(a) add fictitious unitary intervals

(b) the meeting graph

Figure 2.9: The Meeting Graph

Since there are some values which are alive during several iterations of the kernels, these values interfere with themselves because every h steps a new value is defined by the statement. In this case, we have to unroll the motif in order to be able to explicitly allocate distinct registers to distinct values by coloring the circular interval graph.

Theorem 2.2 *[ELM95] Let $M_t = (V_{R,t}, E_M, \omega)$ be the meeting graph of a circular graph $C_h(G)$ with a width w. Let $\mathcal{D}$ be the set of all possible decompositions of M_t into circuits ($D_i \in \mathcal{D}$, $D_i = \{C_{i_1}, \ldots, C_{i_n}\}$). Then, the minimal unrolling degree of the motif necessary to obtain an optimal allocation with w registers is:*

$$u(M_t) = \min_{D_i \in \mathcal{D}} lcm(\rho_{i_1}, \ldots, \rho_{i_n})$$

in which ρ_{i_j} is the number of turns around circle of the circuit C_{i_j}:

$$\rho_{i_j} = \frac{\sum_{u^t \in C_{i_j}} \omega(u^t)}{h}$$

and lcm denotes the least common multiple.

The width of a circuit in the meeting graph is equal to the number of turns around the circle because we ensure that the width is constant by inserting fictitious unitary intervals. Computing the set of all possible decompositions into circuits is NP-complete. So the purpose is to write an algorithm which looks for a "good" decomposition, i.e. the one which reduces the unrolling degree. Assuming such a decomposition, the following theorem defines the unrolling degree necessary for coloring the circular interval graph.

Theorem 2.3 *[Lel96] Let $M_t = (V_{R,t}, E_M, \omega)$ be the meeting graph of a circular graph $C_h(G)$ with a width w. Assume that M_t is composed of n elementary circuits $(C_1, \ldots, C_n)$ with their corresponding width $(\rho_{C_1}, \ldots, \rho_{C_n})$. Then, the corresponding loop can be allocated with w registers if we unroll the motif by the following degree:*

$$lcm(\rho_{C_1}, \ldots, \rho_{C_n})$$

For instance, if we decompose the meeting graph of Figure 2.9 into two elementary circuits $C_1 = (v_1, I_1, I_4, v_1)$ and $C_2 = (v_3, I_2, I_3, v_2, I_5, v_3)$, we need to unroll the loop $lcm(1, 3) = 3$ times. We then allocate one register for C_1 and 3 registers for C_2.

After unrolling the loop, a cyclic register allocation with w available registers is done by coloring the corresponding circular interval graph with w colors. For this purpose, we have to find a decomposition of the unrolled meeting graph into w elementary circuits.

Theorem 2.4 *[Lel96] Let $M_t = (V_{R,t}, E_M, \omega)$ be the meeting graph of a circular graph $C_h(G)$ with a width w. A register allocation of the modulo scheduled loop G is exactly a decomposition of M_t into w elementary circuits.*

The authors suggest to look for circuits with small costs if we want to reduce the unrolling degree. This latter constitutes a hard problem, because reducing a *lcm* isn't a linear problem. Some architectures, as Cydra 5 and IA 64 offer architectural support for register allocation of SWP loops. Next section shows how such a feature can be utilized.

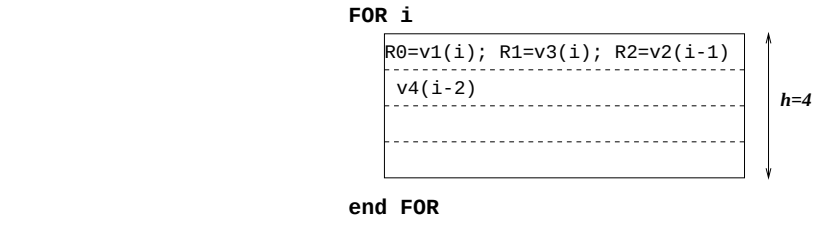
2.4.1 Rotating Register File

A rotating register file (RRF) [DHB89, DT93, RLTS92, SRM94] is a hardware feature to prevent successive lifetimes intervals from being assigned to the same *physical* registers. Conventional registers are accessed using absolute addresses, e.g. register number 3. Nonetheless, in a RRF,

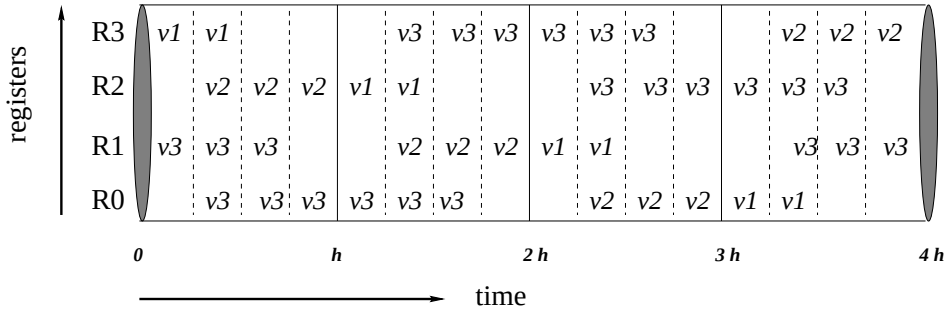
a register number k specified in a statement addresses the physical register $(RRB + k)$ *mod* s, where RRB is a *rotating register base* and s the number of physical registers. At the end of each kernel (special branch instruction), RRB is decremented so that the same register accessed in next iteration is named $(RRB - 1 + k)$ *mod* s. The compiler must take into account this behavior by generating an adapted code. For instance, assuming 4 physical registers, the compiler must be aware that a value written in the architectural register 0 (physical register 0) must be accessed from the architectural register 2 (physical register 0) two kernels latter.

Thanks to RRF, we do not need to unroll the loop. We can always find a cyclic register allocation with at most $w + 1$ registers if the size of the register file is $s \geq w + 1$.

Theorem 2.5 *[Lel96] A loop can be allocated on a rotating register file of size s if there exists a hamiltonian circuit C in the meeting graph with a width* $\rho(C) \leq s$



(a) Allocated Loop kernel



(b) Distinct Values in the RRF

Figure 2.10: Register Allocation in a Rotating Register File

The meeting graph of Figure 2.9 has the hamiltonian circuit $C = (v_1, I_1, I_2, I_3, v_3, I_4, I_5, v_2, v_1)$. We can allocate the values v_1, v_2 and v_3 to a rotating register file of size 4. We allocate these values in the same order that they appear in the hamiltonian circuit. Figure 2.10.(a) shows the generated code (loop kernel) with register allocation. Part (b) shows the distinct values in the RRF : for instance, the value v_3 does not interfere with itself because it is written on a distinct physical register every $h = 4$ steps.

If there is no hamiltonian circuit, we can always create one by adding a complete turn of unitary fictitious intervals in the meeting graph. If no hamiltonian circuit exists in the MG, we can guarantee that there is no a cyclic register allocation with MAXLIVE registers on a RRF [Lel96]. We have to use one extra register, which yields to allocate MAXLIVE+1 registers. One of intrinsic reasons is that the RRF simulates "shifting" actions to move values within physical registers. Depending on the SWP schedule, we may need one extra register to complete this circular moving, since we need 3 registers to permute two values between two distinct registers. This problem arises particularly for superscalar codes. Since we cannot express statically the parallelism between operations, two lifetime intervals cannot meet and, thus, are serialized in the generated code. Consequently, we may need one extra register to cyclically permute all the values in registers.

Proposition 2.2 *[Lel96] There always exists a hamiltonian circuit in the meeting graph of a software pipelined loop if we add a tour of unitary fictitious circular intervals.*

Therefore, a sufficient condition for allocating MAXLIVE+1 registers on a RRF for a software pipelined loop arises:

Theorem 2.6 *[Lel96] Let $M_t = (V_{R,t}, E_M, \omega)$ be the meeting graph of a circular graph $C_h(G)$ with a width w. It is always possible to allocate registers to the loop G in a rotating register file with at least $w+1$ registers.*

Before concluding this chapter, we would like to introduce a loop transformation called retiming. This transformation, as we will see, allows to solve some of the problems in this thesis.

Retiming Transformation

Retiming [LS91] (also called loop shifting [DH00]) consists of the following graph transformation: for each statement u, we associate a shift $r(u)$ which means that we delay the operation $u(i)$ by $r(u)$ iterations. Basically, we only change the column numbers. Then, each statement u which was representing the operations of the form $u(i)$ represents now the operations of the form $u(i - r(u))$. The new distance of each arc $e = (u, v)$ becomes $\lambda_r(e) = \lambda(e) + r(v) - r(u)$ since the dependence is from $u(i - r(u))$ to $v(i - r(v) + \lambda(e))$. Then, we have a one-to-one correspondence between the schedules of the original loop and the schedules of the retimed one. σ_r is a schedule for the retimed graph iff the function σ defined by $\sigma(u(i)) = \sigma_r(u(i + r(v)))$ is a schedule for the original DDG.

Consequently, a retiming does not change the sum of the distances in any circuit, nor the sum of its delays, while preserving the same problem (loop). Indeed, the retimed graph is only another representation of the loop.

Note that a retiming is called valid if all the distances of the transformed graph are positive. Finding a valid retiming (from a non valid one) is a polynomial problem [LS91]. Figure 2.11 gives an illustration. If we use the shifts of Part (b) to apply a retiming on the graph of Part (b), we obtain the retimed graph of Part (c).

2.5 Conclusion

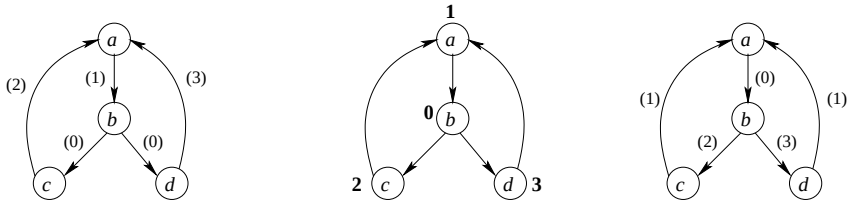
This chapter has introduced our hypothesis about the generic ILP architecture and has defined some important terms that we use in this part of the thesis. Circular register need is defined

by circular intervals. An integer programming model with reduced constraints matrix size is provided and is used in next chapters to analyze register pressure.

While local register allocation of already scheduled DAGs is easy, cyclic register allocation of modulo scheduled loops is slightly different. Since lifetime intervals are circular, some statements may produce interfering values inside the motif. We must unroll the loop to explicitly address these distinct values and to allocate them to different registers. A theoretical framework, called meeting graph [ELM95, ELM97, dWELM99, Lel96], formulates the exact unrolling degree depending on a circuit decomposition of MG.

Optimizing the unrolling degree is a difficult task. A hardware feature, called rotating register file, allows to avoid unrolling the kernel. A sufficient condition for cyclic register allocation with MAXLIVE registers on a RRF is the existence of a hamiltonian circuit in the MG. If it does not exist, we can create it by using one extra register.

Next chapter studies cyclic RS devoted to keep register pressure under control before SWP scheduling.



(a) Original Graph (b) Shift Coefficients (c) Retimed Graph

Figure 2.11: Valid Graph Retiming

Chapter 3

Cyclic Register Saturation

This chapter describes our work on cyclic register saturation (CRS) [TE02]. We provide algorithms and intLP models to check if register constraints are obsolete (satisfied) before scheduling. As in the acyclic case, this problem is NP-complete. We show how we handle the NP-hard problem of reducing CRS under the limit of available registers. This chapter improves and overcomes drawbacks of our previous work on loops in [TT00].

This chapter is organized as follows. Section 3.1 shows how we compute CRS. We provide an exact formulation based on integer programming. We also present an approximative method that decomposes the problem into two parts. The first part, based on integer programming, looks for a valid retiming that maximizes the interferences of distinct instances of the same statement. The second part, based on algorithmic approximation, maximizes the interferences of distinct statements. Section 3.2 studies the problem of CRS reduction under a fixed critical circuit. Before concluding, we show some experiments in Section 3.3.

Let $G = (V, E, \delta, \lambda)$ be a loop. The cyclic register saturation (CRS) is the maximal register requirement for all valid software pipelined schedules :

$$CRS_t(G) = \max_{\sigma \in \Sigma(G)} CRN_t^{\sigma}(G)$$

where $CRN_t^{\sigma}(G)$ is the cyclic register need of type f for the SWP schedule σ. A software pipelined schedule which needs the maximum number of registers is called a *saturating SWP schedule*. The excessive values (maximum values simultaneously alive) in a saturating schedule are called *saturating values*.

Theorem 3.1 *Computing the cyclic register saturation of a register type $t \in \mathcal{T}$ is NP-complete.*

Proof :

The proof becomes obvious by reducing to the the acyclic case [Tou01d, Tou01c, Tou01a]. To fix the ideas, we take the class of the data dependence graphs as described in Figure 3.1 : there are not inter-iteration flow arcs of the register considered type (no flow loop carried dependences)

$$\forall e \in E_{R,t} : \qquad \lambda(e) = 0$$

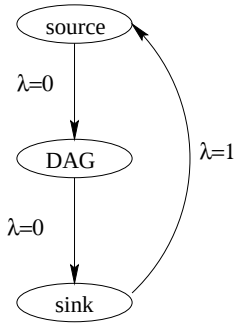


Figure 3.1: Class of Loops with no Possible Iteration Overlapping

and $MII = L$, i.e. any modulo schedule cannot overlap the iterations. Any motif only consists of one iteration of the original loop. So, the cyclic register need is strictly defined by the interference between the values inside the original loop body, which is the maximal register need for the acyclic schedules of the loop body. This problem has been proven NP-complete in [TT00].

⌟

The next section presents how we compute CRS.

3.1 Computing Cyclic Register Saturation

3.1.1 Exact Formulation of CRS Computation

Before computing the exact CRS, we must first note that the exact maximal register requirement may not exist. This is, for instance, the case for acyclic DDGs ($MII = 0$). We can theoretically schedule all the values of all iterations in parallel. Assuming infinite number of iterations, this may lead, in theory, to an infinite number of values simultaneously alive. As mentioned in Chapter 2, infinite ILP degree is not considered in SWP since we focus on building a kernel (software pipelined loop). Then, we set $MII \geq 1$.

Furthermore, if we do not bound L, the total schedule time of one original iteration, then the maximal number of parallel iterations ($\lceil L/MII \rceil$) may be infinite. In other words, even if we set $MII \geq 1$, the exact maximal register requirement may not exist if we do not bound L.

For these reasons, we bound our problem by computing the cyclic register saturation of a subset $\Sigma_L(G) \subseteq \Sigma(G)$. That is, we compute the maximal register requirement for all valid software pipelined schedules with the property that the total schedule time of one iteration does not exceed L. This is appropriate for us, since the domain set of variables must be bounded in our intLP formulation.

The exact formulation of CRS computation is derived from Section 2.3.1:

$$\text{Maximize} \sum_{\text{acyclic in\_fraction\_of\_}h \text{ interval } I} x_I + \sum_{u^t \in V_{R,t}} p_{u^t}$$

subject to the variables and constraints defined in Section 2.3.1.

The size of the model is $\mathcal{O}(|V_{R,t}|^2)$ variables and $\mathcal{O}(|E|+|V_{R,t}|^2)$ constraints (Section 2.3.1 on page 20). The coefficients of the constraints matrix are all bounded by $\pm L \times \lambda_{max} h$, where λ_{max} is the maximal dependence distance in the loop. To compute CRS, we iterate the initiation interval h from h_0 to $h_{max} = L$, i.e. we create a model for each $h \in [h_0, L]$. The register saturation is the maximal solution of all these models. This method may involve to solve too many intLP models. However, we can consider a tight upper-bound. The following corollary states that instantiating a model for only $h = L$ while relaxing the upper-bound L' is sufficient to compute CRS. Let us start by the following lemma.

Lemma 3.1 *Let $G = (V, E, \delta, \lambda)$ be a DDG of a loop. The maximal register requirement of all the software pipelined schedules $\sigma([rn], [cn], h)$ with an initiation interval $h_0 \leq h \leq L$ is less or equal to the maximal register requirement of all the software pipelined schedules with an initiation interval $h' = h + 1$ with $L' = L + 1 + \lfloor L/h \rfloor$. Formally:*

$$\max_{\substack{\sigma([rn],[cn],h) \in \Sigma_L(G) \\ h_0 \leq h \leq L}} CRN_t^{\sigma}(G) \leq \max_{\sigma'([rn'],[cn'],h+1) \in \Sigma_{L+1+\lfloor L/h \rfloor}(G)} CRN_t^{\sigma'}(G)$$

Proof:

It is a direct consequence of Proposition 2.1 on page 17. If we increment h by one, we have to increment L by $1 + \lfloor L/h \rfloor$ to get at least one valid software pipelined schedule with the same register requirement:

$$\forall \sigma([rn], [cn], h) \in \Sigma_L(G)/h \leq L, \quad \exists \sigma'([rn'], [cn'], h+1) \in \Sigma_{L+1+\lfloor L/h \rfloor}(G) :$$

$$CRN_t^{\sigma'}(G) \geq CRN_t^{\sigma}(G)$$

⌟

Corollary 3.1 *Let $G = (V, E, \delta, \lambda)$ be a DDG of a loop. Then, the exact CRS of G assuming L as an upper-bound of the total schedule time of one iteration is lower or equal to the maximal register requirement with $h = L$ if we relax the upper-bound $L' \geq L$. Formally:*

$$\max_{\sigma([rn],[cn],h_0 \leq h \leq L) \in \Sigma_L(G)} CRN_t^{\sigma}(G) \leq \max_{\sigma([rn],[cn],L) \in \Sigma_{L'}(G)} CRN_t^{\sigma}(G)$$

where L' is the $(L - h_0)^{th}$ term of the following recurrent sequence ($L' = U_L$):

$$\begin{cases} U_{h_0} &= L \\ U_{h+1} &= U_h + 1 + \lfloor U_h/h \rfloor \end{cases}$$

Proof:

It is a direct consequence of Lemma 3.1:

$$\max_{\sigma([rn],[cn],h)\in\Sigma_L(G)} CRN_t^\sigma(G) \leq \max_{\sigma([rn],[cn],h+1)\in\Sigma_{L+1+\lfloor L/h\rfloor}(G)} CRN_t^\sigma(G) \leq ...$$

$$... \leq \max_{\sigma([rn],[cn],L)\in\Sigma_{U_{L-h}=U_{L-h-1}+1+\lfloor U_{L-h-1}/(L-h-1)\rfloor}(G)} CRN_t^\sigma(G)$$

That is, we relax the upper-bound L at each step, from h to L, i.e. $(L-h)$ times. Since CRS is defined for all initiation intervals, starting from $h = h_0$ amounts to relax the upper-bound $(L-h_0)$ times, as follows.

$$\max_{\sigma([rn],[cn],h_0)\in\Sigma_{L=U_{h_0}}(G)} CRN_t^\sigma(G) \leq \max_{\sigma([rn],[cn],h_0+1)\in\Sigma_{U_{h_0+1}=L+1+\lfloor L/h\rfloor}(G)} CRN_t^\sigma(G) \leq ...$$

$$... \leq \max_{\sigma([rn],[cn],L)\in\Sigma_{U_L=U_{L-1}+1+\lfloor U_{L-1}/(L-1)\rfloor}(G)} CRN_t^\sigma(G)$$

⌟

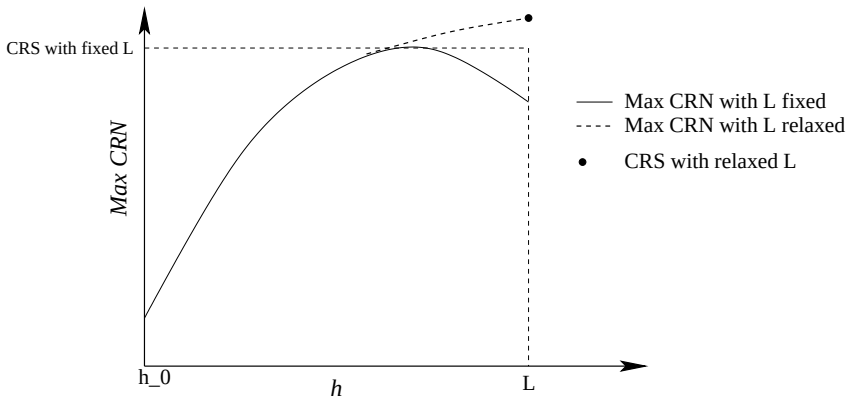


Figure 3.2: Maximal Cyclic Register Need vs. Initiation Intervals

This corollary enables us to only solve intLP systems with $h = L$; the upper-bound L' must be relaxed. The computed CRS is greater than or equal to the optimal CRS by assuming L as an upper-bound of the total schedule time of one iteration. Figure 3.2 draws theoretical asymptotic curves to explain the meanings of Corollary 3.1. If we fix L as an upper-bound of the total schedule time of one iteration, the maximal register requirement under a fixed execution rate h may not be an increasing function of h. At a certain value of h, the maximal register requirement may decrease if the upper-bound is not relaxed.

Next section investigates a heuristics to approximate the cyclic register saturation. It combines integer programming and a pure algorithmic solution.

3.1.2 A FCLR Heuristic: First Columns Last Rows

In this section, we present a First-Columns-Last-Rows heuristics which constructs a SWP motif in order to approximate a saturating schedule in terms of cyclic register need. Our heuristics consists of two main steps:

1. We first find column numbers that maximize the number of iterations traversed by values. This intends to maximize the number of copies of the values (turns around the circle).

2. Once we compute column numbers, we can build the DAG of the motif to find row numbers that guarantee inter-motif dependences. We must maximize interferences between lifetime intervals inside this motif. For this purpose, we use the DAG technique studied in [TT00, Tou01e] to construct an acyclic saturating schedule.

Maximizing the Traversed Motifs (Column Numbers)

A value does not span the motif if it is defined and consumed inside the same motif. If it is consumed i motifs later, then it spans $i+1$ motif and there are at most $i+1$ copies of this values which interfere with each other. Given a software pipelined schedule, the number of the motifs traversed by a value u^t to be consumed by an operation v is equal to $\big(cn(v) + \max_{e=(u,v)\in E_{R,t}} \lambda(e)\big) - cn(u)$. The total number of motifs spanned by a value u^t is then:

$$s_{u^t} = \max_{v \in Cons(u^t)} \big(cn(v) + \max_{e=(u,v)\in E_{R,t}} \lambda(e)\big) - cn(u)$$

The number of motifs crossed by a value is not exactly the number of its copies (turns). For instance, the value v_3 in Figure 2.2 on page 11 spans 3 successive motifs but has only one copy (complete turn). The general relation between the number of copies and the number of traversed motifs can be stated as:

$$p_{u^t} \geq s_{u^t} - 2$$

In our heuristics, we want to maximize the number of copies of all values. So, we have to find column numbers that maximize the number of traversed motifs. This is done by considering the following linear programming model:

- Maximize
$$\sum_{u^t \in V_{R,t}} s_{u^t}$$

- Subject to:

  - the column numbers must be valid, i.e. there exists at least one software pipelined schedule with the computed column numbers [Saw97]:
$$\forall e = (u,v) \in E : \qquad cn(v) - cn(u) \geq -\lambda(e)$$
    which is equivalent to finding a valid retiming ($r(u) = cn(u)$).
  - we bound the column numbers according to L:
$$\forall u \in V : \qquad cn(v) \leq \overline{cn}(v)$$

– the number of traversed motifs by a value is:

$$\forall u \in V_{R,t} \ : \qquad s_{u^t} = \max_{v \in Cons(u^t)} \Big(cn(v) + \max_{e=(u,v) \in E_{R,t}} \lambda(e) \Big) - cn(u)$$

We use the linear constraints of the "maximum" defined [Tou01d, Tou01c, Tou01a].

The size of this model is bounded by $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E| + |V|^2)$ constraints. Unfortunately, we do not have an algorithmic solution for this problem, nor do we have a proof for its computational complexity. We propose to use a heuristics to solve this intLP.

Remark In this section, we have maximized the number of traversed motifs by assuming that a statement u defines its value u^t during the current kernel. In fact, this may not be correct depending on the row number and the write delay in the register of this statement: we can choose a row number and an initiation interval for u in such a way that the statement u of the current motif defines its value in a further motif. This is because the definition of u^t proceeds $\delta_{w,t}(u)$ after $rn(u)$ and may cross the h barrier. For instance, consider the SWP kernel of Figure 3.3 in which the value produced by a is consumed one kernel later by d. According to our assumption, the value traverses one motif. However, by considering the writing latency, the initiation interval and the row number of a, this value is defined during the next kernel, and hence does not cross a motif. This problem will be fixed in the next section when we compute row numbers and suitable initiation interval.

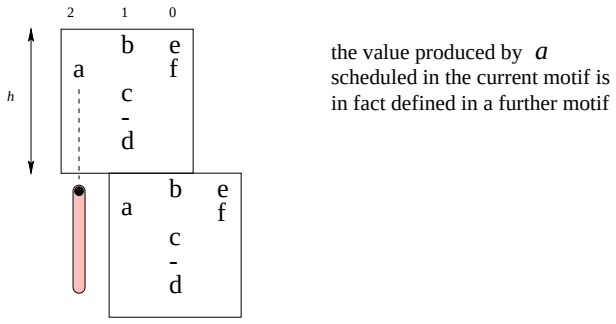


Figure 3.3: Value Definition in a Further Motif

Maximizing the Interferences inside the Motif (Row Numbers)

After determining column numbers in the first step, we have possibly maximized the number of copies. In this second phase, we define the row numbers of statements in such a way that interferences between (in_fraction_of_h) lifetime intervals inside the motif are maximized.

After fixing column numbers, some dependences become inter-motif, i.e. they involve operations from different motifs and hence are satisfied by the successive execution of iterations (see Figure 3.4). However, other dependences become intra-motif, i.e. they involve operations inside the same motif: determining row numbers is constrained by these dependences. For this purpose, we build a DAG from the original DDG G which contains the set of statements and the set of intra-motif dependences:

1. the inter-motif dependences are the set of arcs[Saw97]

$$E_1 = \{e = (u, v) \in E \,/\, cn(v) - cn(u) > -\lambda(e)\}$$

these dependences are satisfied by the successive execution of the motif. So, we evict from G all arcs that belong to E_1 ;

2. the intra-motif dependences are the set of arcs

$$E_2 = \{e = (u, v) \in E \,/\, cn(v) - cn(u) = -\lambda(e)\}$$

these dependences must be ensured by row numbers. So, we keep these arcs to build our inner-motif dependence DAG.

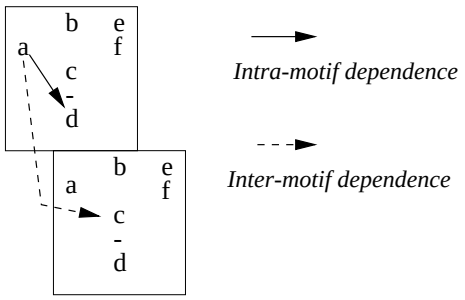


Figure 3.4: Inter and Intra Motif Dependences

Once the DAG is built, we use our DAG technique ([TT00, Tou01e]) to keep as many values alive as possible inside the motif. However, there are some values entering the motif (values produced backwards from precedent kernels) and some others exiting it (values produced for forward motifs)[1]. These virtual values must be inserted into the DAG as follows.

1. We insert a virtual value u^t_{entry} iff $\exists e = (u, v) \in E_{R,t} \cap E_1$, i.e. if an operation v consumes a value u^t produced by previous motifs. To model the fact that these entry values are alive from the top of the motif, we add serial arcs from u^t_{entry} to the sources of the DAG with a latency 0. We insert a flow arc from u^t_{entry} to each v such that $e = (u, v) \in E_{R,t} \cap E_1$ with a latency $\delta_{w,t}(u) + 1$.

2. For all $e = (u, v) \in E_{R,t} \cap E_1$, the value u^t is consumed in a further motif. To model this fact, we add a flow arc from u to the bottom $\perp$ with the latency $\delta_{w,t}(u) + 1$.

Now, we get a DAG with entry and exit values and inter motif dependences. We apply our efficient Greedy-k heuristics to find a saturating acyclic schedule $\bar{\sigma}$. Then, we set :

$$\forall u \in V : \qquad rn(u) = \bar{\sigma}(u)$$

We have determined row and column numbers. We still have to choose a valid value for h. The initiation interval must first ensure the inter-motif dependences. Second, it must fix the problem of traversed motif like noticed in the remark at the end of the previous section (the

[1]These entry and exit values are those which define the lefts and the rights of cyclic lifetime intervals.

fact that a statement must define its value during the current motif). The following initiation interval ensures these two constraints:

$$h = \max_{u \in V} rn(u) + lat(u)$$

Lastly, we have defined a software pipelined schedule σ which maximizes cyclic register need. So, the approximated CRS is:

$$CRS(G) \geq CRS_t^*(G) = CRN_t^{\sigma}(G)$$

Example 3.1.1 *Let us take the DDG shown in Figure 2.1.(a) on page 7. Suppose that the column numbers which maximize the number of spanned motifs are:*

Statement	*cn*	*Traversed Motifs*
v_1	*0*	*1*
v_2	*1*	*4*
v_3	*0*	*2*
v_4	*5*	-

According to these column numbers, there are no intra-motif dependences, i.e. all the dependences are satisfied by inter-motif ones. The DAG built to compute row numbers does not contain any of the original dependences and then statements inside the motif are completely independent. Figure 3.5.(a) shows the DAG after inserting entry and exit values. To find a saturating acyclic schedule for this DAG, we apply our Greedy-k algorithm. This leads to an acyclic schedule which 4 saturating values inside the motif: these values are $v1, v1_{entry}, v2_{entry}, v3_{entry}$. *This acyclic saturating schedule defines the following row numbers:*

Statement	*rn*
v_1	*0*
v_2	*2*
v_3	*2*
v_4	*2*

The initiation interval is set to $h = 5$. *Figure 3.5.(b) shows the circular lifetime intervals in the motif: the width is 8 so the approximated cyclic register saturation is equal to 8. One can remark that the value* v_2 *spans 5 successive motifs but has only 3 copies.*

When $CRS_t(G)$ is $\leq \mathcal{R}_t$, the number of available registers of type t, the DDG G is definitively free from register pressure and can be left unchanged for a further scheduling process. Otherwise, we must reduce it to keep register need under control. However, if $CRS_t^*(G)$ is $\leq \mathcal{R}_t$, then some saturating schedules may still exist since $CRS_t^*(G) \leq CRS_t(G)$. Nevertheless, since CRS_t^* maximizes the cyclic register need, it is very unlikely that a SWP process would require more registers than $CRS_t^*(G)$. In some critical cases, spill code may be introduced, even if $CRS_t^*(G) \leq CRS_t(G)$.

The next section investigates the problem of CRS reduction.

3.2 Reducing Cyclic Register Saturation

This section studies how to add serial arcs to a given DDG $G = (V, E, \delta, \lambda)$ such that its cyclic register saturation of a register type t is limited by a strictly positive integer $\mathcal{R}_t$ under a fixed critical circuit constraint MII. This allows us to guarantee that any software pipelining of the new graph does not require more registers that those available. Consequently, we can always build a valid register allocation without spilling after the SWP process. Note that in the presence of a rotating register file, we have to ensure that the cyclic register saturation does not exceed $\mathcal{R}_t - 1$ registers (consequence of Theorem 2.6 on page 26).

Problem 3.1 (ReduceCRS) *Given a DDG $G = (V, E, \delta, \lambda)$, is there an extended DDG $\overline{G}$ of G such that $CRS_t(\overline{G}) \leq \mathcal{R}_t$ and $MII \leq \overline{MII}$?*

It is clear that the limit $\mathcal{R}_t$ must be greater than or equal to the cyclic register sufficiency (studied in next chapter). Otherwise, there is no solution to this problem and spill code can not be avoided. Unfortunately :

Theorem 3.2 *Reducing the Cyclic Register Saturation is NP-hard.*

Proof :

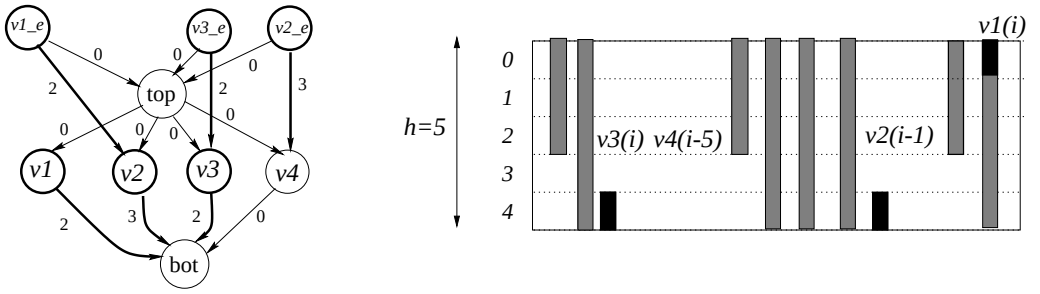
First, ReduceCRS does not belong to NP. Since computing cyclic register saturations is NP-complete (Theorem. 3.1), we cannot check in polynomial time if a given solution $\overline{G}$ has $CRS_t(\overline{G}) \leq \mathcal{R}_t$.

Now, we prove that ReduceCRS can be reduced to the problem of cyclic scheduling under register constraints. Let us start by defining the latter problem.

Definition 3.1 (SRC problem) *Let $G = (V, E, \delta, \lambda)$ be a DDG and $\mathcal{R}_t$ a strictly positive integer. Does it exists a valid schedule $\sigma \in \Sigma_L(G)$ such that :*

$$CRN_t^{\sigma}(G) \leq \mathcal{R}_t \ \wedge \ h \leq \overline{MII}$$

where h is the initiation interval of σ.



(a) the DAG of the motif

(b) saturating cyclic schedule

Figure 3.5: A FCLR Heuristics for Computing the Cyclic Register Saturation

This problem is NP-complete [EGS95].

1. **ReduceCRS $\Longrightarrow$ SRC**

Let $\overline{G}$ be a solution for the ReduceCRS problem. Then, we can build an optimal schedule $\sigma \in \Sigma_L(\overline{G})$ in a polynomial time complexity under only the serial constraints [GS94] with $h = MII \leq \overline{MII}$.

2. **SRC $\Longrightarrow$ ReduceCRS**

Let σ be a solution for SRC, i.e. $CRN_t^\sigma(G) \leq \mathcal{R}_t$ and $h \leq \overline{MII}$. As an example, let us consider the DDG previously shown in Figure 2.1.(a) on page 7 with its corresponding modulo schedule σ in Part (b). That DDG has a register saturation equal to 8 as shown in Figure 3.5 page 37. We want to reduce it to four registers based on the schedule of Figure 2.1.(b) in which the cyclic register requirement is shown in Figure 2.2 page 11.

We have to build an extended DDG $\overline{G}$ such that we guarantee that any software pipelining schedule $\sigma' \in \Sigma(\overline{G})$ produces the same cyclic relative order between values circular lifetime intervals as defined by σ. If a lifetime interval $LT_\sigma(u^t(i))$ is before lifetime interval $LT_\sigma(v^t(i+\alpha))$, then we must guarantee that any software pipelining σ' makes $LT_{\sigma'}(u^t(i))$ before $LT_{\sigma'}(v^t(i+\alpha))$, α is a distance to be defined.

We model the relative cyclic order between circular lifetime intervals by a graph $O = (V_{R,t}, E_\prec, \alpha)$: $e = (u^t, v^t) \in E_\prec$ means that the value produced by $u(i)$ is killed before the definition of the value $v^t(i + \alpha(e))$. $\alpha(e)$ is chosen so that the killing date of $u^t(i)$ must be as close as possible to the definition date of $v^t(i + \alpha(e))$, i.e. both of the two dates must be in a window of size h. Since the schedule times of the distinct instances of the statement v are separated by h clock cycles, there is a unique distance α that defines the cyclic order between $LT_\sigma(u^t(i))$ and $LT_\sigma(v^t(i+\alpha))$ in a window of size h. The constraints that define such distance α between $u^t(i)$ and $v^t(i+\alpha)$ are (u^t not necessarily distinct from v^t):

$$LT_\sigma(u^t(i)) \prec LT_\sigma(v^t(i+\alpha)) \tag{3.1}$$

$$\sigma(v(i+\alpha)) + \delta_{w,t}(v) - k_{u^t(i)} < h \tag{3.2}$$

Since

$$(3.1) \Longleftrightarrow k_{u^t(i)} \leq \sigma(v^t(i+\alpha)) + \delta_{w,t}(v) \Longleftrightarrow k_{u^t} \leq \sigma_v + h \times \alpha + \delta_{w,t}(v)$$

and

$$(3.2) \Longleftrightarrow \sigma_v + h \times \alpha + \delta_{w,t}(v) - k_{u^t} < h$$

(3.1) and (3.2) amount to:

$$0 \leq \sigma_v + h \times \alpha + \delta_{w,t}(v) - k_{u^t} < h$$

Then, α is the unique integer that belongs to the interval:

$$\frac{k_{u^t} - \sigma_v - \delta_{w,t}(v)}{h} \leq \alpha < 1 + \frac{k_{u^t} - \sigma_v - \delta_{w,t}(v)}{h}$$

$$\Longrightarrow \alpha = \left\lceil \frac{k_{u^t} - \sigma_v - \delta_{w,t}(v)}{h} \right\rceil$$

Now, we have completely defined the cyclic ordering graph $O = (V_{R,t}, E_{\prec}, \alpha)$. Note that the arcs $E_{\prec}$ are defined from each values u^t to another v^t (u^t not necessarily distinct from v^t), since a periodic schedule makes circular all the lifetime intervals: for any $(u^t, v^t) \in V^2_{R,t}$, there exists a unique α (under the constraints just defined above) such that $LT_\sigma(u^t(i)) \prec LT_\sigma(v^t(i+\alpha))$. As an illustration, Figure 3.6.(a) shows the cyclic relative ordering between the values deduced from Figure 2.2 on page 11. For instance, $LT_\sigma(v_2(i)) \prec LT_\sigma(v_1(i+2))$, thus there is a cyclic ordering arc $e = (v_2, v_1)$ in Figure 3.6.(a) with $\alpha(e) = 2$. Also, $LT_\sigma(v_1(i)) \prec LT_\sigma(v_1(i+1))$, thus there is a cyclic ordering arc $e = (v_1, v_1)$ in Figure 3.6.(a) with $\alpha(e) = 1$.

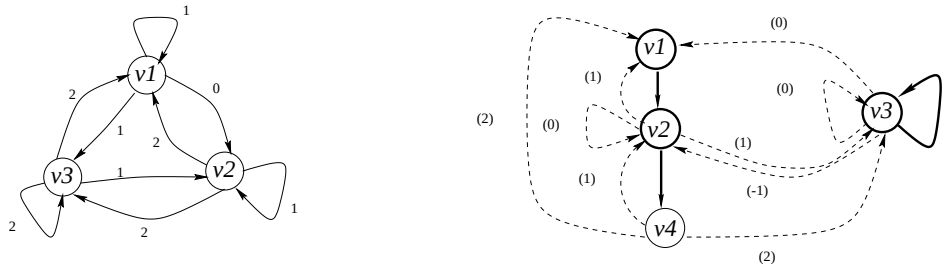


Figure 3.6: Cyclic Ordering

Now, let us see how to build an extended DDG $\overline{G}$ based on this cyclic ordering, i.e. how to report cyclic precedence relations between values lifetime intervals. For each order $e = (u, v) \in E_{\prec}$ between two values u^t and v^t, we must guarantee that the killing date of u^t is always performed before the definition date of $v(i + \alpha(e))$:

$$k_{u^t} \leq \sigma\left(v(i + \alpha(e))\right) + \delta_{w,t}(v)$$

This means that $\forall u' \in Cons(u^t)$:

$$\sigma\left(u'(i + \lambda\left((u, u')\right)\right) + \delta_{r,t}(u') \leq \sigma\left(v(i + \alpha(e))\right) + \delta_{w,t}(v)$$

$$\Longleftrightarrow \sigma\left(u'(i)\right) + \delta_{r,t}(u') - \delta_{w,t}(v) \leq \sigma\left(v(i + \alpha(e) - \lambda\left((u, u')\right)\right)$$

in which $\lambda\left((u, u')\right)$ is the distance of the flow dependence between u and its consumer u'. This is done by adding a serial arc e' to G from each consumer $u' \in Cons(u^t)$ to v with:

$$\delta(e') = \delta_{r,t}(u') - \delta_{w,t}(v) \qquad \text{and} \qquad \lambda(e') = \alpha(e) - \lambda\left((u, u')\right)$$

Figure 3.6.(b) is the extended graph which reduces register saturation to 4. In that figure, the added serial arcs appear with dashed lines and tagged with only the distances. As an example, there is an order between v_1 and v_3 with a distance $\alpha = 1$. Since v_2 consumes v_1 with distance $\lambda = 0$, we add a serial arc from v_2 to v_2 with

a distance $\alpha - \lambda = 1$. Note that some added serial arcs may be redundant. As an illustration, there is an order between v_3 and itself with a distance $\alpha = 2$. Since v_3 consumes itself with a distance $\lambda = 2$, this produces a serial arc in G from v_3 to itself with $\alpha - \lambda = 0$. This serial arc is always satisfied by any schedule and can be removed from $\overline{G}$.

By adding all these serial arcs, we build an extended DDG $\overline{G}$ that has the following characteristics.

- Any software pipelined schedule σ' of $\overline{G}$ produces a circular order between circular lifetime intervals as defined by σ. So, σ' cannot need more registers than σ. This is because if two lifetime intervals do not interfere with each other according to σ , they cannot interfere with each other according to σ'.
  1. The number of distinct interfering copies (turns around the circle) of each statements u with σ' cannot exceed the number p_{u^t} of distinct interfering copies with σ. This is because we have according to σ $LT_\sigma(u^t(i)) \prec LT_\sigma(v^t(i + p_{u^t} + 1))$. Since we report the cyclic order $e = (u^t, u^t)$ with $\alpha(e) = p_{u^t} + 1$ in the extended DDG $\overline{G}$, at most p_{u^t} copies of u^t may interfere according to a schedule σ' of $\overline{G}$.
  2. The in_fraction_of_h intervals inside the motif are constrained to satisfy the same precedence relation as defined by σ. If two in_fraction_of_h intervals (l, r) and (l', r') do not interfere with each other according to σ, then they cannot interfere according to σ'. Otherwise it means that σ' violates one of the added serial arcs.
- σ is a valid software pipelined schedule for $\overline{G}$ since it satisfies all the introduced serial arcs. Then, any introduced circuit by the serial arcs has a positive distance. If its distance is null, then its latency is necessarily negative or null (the DDG remains valid) :

$$\forall \text{ circuit } C \in \overline{G} : \qquad \lambda(C) \geq 0 \wedge \big(\lambda(C) = 0 \Longrightarrow \delta(C) \leq 0\big)$$

- Since the initiation interval h of σ is lower than or equal to $\overline{MII}$, a possible introduced critical circuit in $\overline{G}$ is not greater than $\overline{MII}$. Otherwise it means that σ isn't a valid software pipelined schedule for $\overline{G}$.

From above, we deduce :

$$\forall \overline{\sigma} \in \Sigma_L(\overline{G}) \qquad CRN_t^{\overline{\sigma}}(\overline{G}) \leq CRN_t^{\sigma}(G)$$

and hence

$$CRS_t(\overline{G}) \leq CRN_t^{\sigma}(G) \leq \mathcal{R}_t$$

⌟

From the previous proof, we deduce that reducing the cyclic register saturation is equivalent to finding a software pipelined schedule with a minimal initiation interval which does not require more than R_t registers. Our exact formulation uses the intLP system that computes MAXLIVE,

previously defined in Section 2.3.1 on page 20. Since we express in the latter formulation the exact register requirement, we only have to bound the register requirement of each register type:

$$\forall t \in \mathcal{T} : \sum_{\text{acyclic in\_fraction\_of\_}h \text{ interval } I} x_I + \sum_{u^t \in V_{R,t}} p_{u^t} \leq \mathcal{R}_t$$

Solving this intLP system yields to two cases.

1. If a feasible solution is found, then there exists a software pipelined schedule σ such that $CRN_t^\sigma(G) \leq \mathcal{R}_t$. Then, we add serial arcs to the DDG as described in the previous proof. The critical circuit of the extended DDG is lower than or equal to h.

2. If no solution exists, then a software pipelined schedule of initiation h such that $CRN_t^\sigma(G) \leq \mathcal{R}_t$ does not exist. We cannot reduce cyclic register saturation with respect to the critical circuit $MII \leq h$. We have to increment h (in dichotomic way between $h_{min} = h$ and $h_{max} = L$) until reaching a solution or not. If no solution exists, spill code must be introduced.

The complexity of the intLP system is bounded by $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E| + |V|^2)$ constraints.

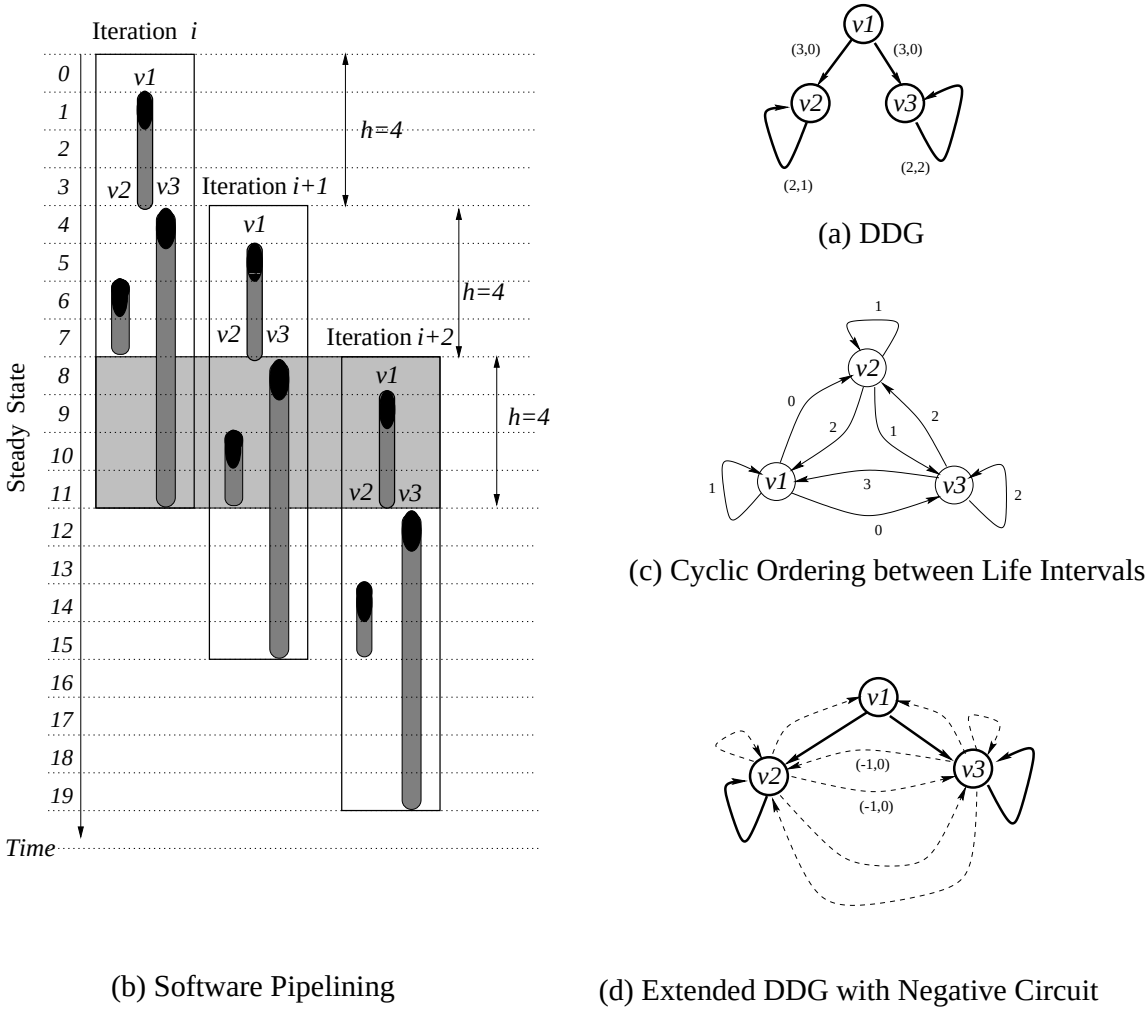
However, as in case of DAGs ([Tou01d]), an optimal solution may need to introduce a circuit C with a null distance $\lambda(C) = 0$ and a negative latency $\delta(C) \leq 0$. The next section discusses this problem.

3.2.1 Problem of Circuits with Null Distances

This section explains why a circuit C with a null distance $\lambda(C) = 0$ and a negative latency $\delta(C) \leq 0$ is a problem, even if from the theoretical perspective, there exists a modulo schedule also in presence of these kinds of circuits in the DDG.

We must remind that the purpose of register saturation analysis is to proceed by ensuring in the first steps of compilation that any schedule of a given DDG would not require more registers than those available. The scheduling phase is mainly constrained by resources (functional units or other rules) of the target architecture. If the extended DDG produced by the register saturation reduction contains a circuit with a null distance and negative or null latency, we cannot guarantee the existence of a software pipelined schedule under resource constraints. This is because the negative latency introduces scheduling constraints of types "not later than" which may not be satisfied in the presence of resource constraints.

A sufficient condition so that these circuits are present is if σ enforces the fact that more than one consumer on the same iteration of a value u does not interfere with u. In this case, a negative cycle is introduced to ensure that no one of the consumers interfere with the value u (and the fact that these consumers belongs to the same iterations makes the distance of the circuit null). For instance, let us consider the DDG of Figure 3.7.(a). A schedule which requires four registers is presented in part (b). We see that the two consumers v_2 and v_3 of the value v_1 are in the same iteration (the distance of the dependence between v_1 and his two consumers is



(a) DDG

(b) Software Pipelining

(c) Cyclic Ordering between Life Intervals

(d) Extended DDG with Negative Circuit

Figure 3.7: Null Circuits with Negative Latency

null) ; the schedule makes both of lifetime intervals v_2 and v_3 ordered after v_1 as shown in the cyclic ordering graph in part (c). To guarantee this cyclic ordering, we extend the initial DDG with the serial arcs as shown in part (d) : here, we see that there is a circuit from v_2 to v_3 with a null distance and negative latency. In the presence of resource constraints, we may not be able to find a valid software pipelined schedule which satisfies this circuit.

A first solution to this problem is to not introduce serial arcs with negative or null latencies. This method does not change the intLP system, since we only have to set $\delta_{r,t}(u) = 0$ and $\delta_{w,t}(u) = -1$ for each statement u. Furthermore, it does not alter the optimality of the solution in the case of sequential codes (superscalar), but may do so in static issue codes (VLIW). An optimal solution is explained below.

The problem of negative circuits is overcame by guaranteeing the existence of a topological sort of the loop body. However, since the constructed DDG may contain negative distances, we may not be able to detect some circuits with null distances. We have then to retime the graph so as to have only positive distances[2]. Then, each arc with a null distance in the retimed graph is an arc in the loop body. If we guarantee that there is no null distance circuit in the retimed graph, then the non retimed DDG does not contain a null circuit (and vice versa). For this purpose, we modify the intLP system by adding the following variables and constraints for retiming simulation.

- We add the following variables.
  - We add new topological sort variables. For each statement $u \in V$, we define an integer variable d_u.
  - We add new retiming shifts variables. For each statement $u \in V$, we define an integer variable r_u.
  - For each couple of values (u^t, v^t), we add a new variable $\alpha^t_{u,v}$ so as to compute the distance of an introduced serial arc $e = (u', v) \ / \ u' \in Cons(u^t)$.
- In the previous intLP system, we replace each distance $\lambda(e) \ / e = (u, v)$ by $(\lambda(e) + r_v - r_u)$. Since h is constant in the intLP, our system remains linear.
- We add the following constraints.
  - If the lifetime interval of a value $u(i)$ precedes the lifetime interval of a value $v(i+\alpha)$, then we introduce a serial arc from each consumer $u' \in Cons(u^t) \ / \ e = (u, u') \in V_{R,t}$ to v with a distance $(\alpha - \lambda(e))$, as explained in the previous proof. In the retimed graph, this distance becomes $\alpha - \lambda(e) + r_v - r_{u'}$. In order to compute α, we write the following constraints.
    - The lifetime interval of a value $u(i)$ must precede the lifetime interval of a value $v(i + \alpha)$. Then, we must have in the retimed graph :

$$\forall u, v \in V_{R,t} \ : \ k_{u^t} \leq \sigma_v + \delta_{w,t}(v) + h \times (\alpha^t_{u,v} + r_v - r_u)$$

[2]Retiming is possible because the DDG does not contain a circuit C with negative distance $\lambda(C) < 0$

* The cyclic ordering of lifetime intervals is defined in a window $[0, h[$. That is, the definition date of $v(i+\alpha)$ and the killing date of $u(i)$ must be inside a motif size. Then, we must have in the retimed graph :

$$\forall u, v \in V_{R,t} \; : \; \sigma_v + \delta_{w,t}(v) + h \times (\alpha^t_{u,v} + r_v - r_u) - k_{u^t} < h$$

– We have to guarantee the existence of a topological sort of the retimed loop body.

* We write the bounding constraints :

$$\forall u \in V \; : \quad d_u \leq |V|$$

* For original arcs, we write :

$$\forall e = (u, v) \in E \; : \; \lambda(e) + r_v - r_u = 0 \Longrightarrow d_u < d_v$$

* For introduced serial arcs : $\forall u, v \in V_{R,t}$,

$$\forall u' \in Cons(u^t) \,/\, e = (u, u') \in V_{R,t} \; : \quad \alpha^t_{u,v} - \lambda(e) + r_v - r_{u'} = 0 \Longrightarrow d_{u'} < d_v$$

– The retiming must be valid.

* For each original arc :

$$\forall e = (u, v) \in E \; : \quad \lambda(e) + r_v - r_u \geq 0$$

* For introduced serial arcs : $\forall u, v \in V_{R,t}$,

$$\forall u' \in Cons(u^t) \,/\, e = (u, u') \in V_{R,t} \; : \; \alpha^t_{u,v} - \lambda(e) + r_v - r_{u'} \geq 0$$

There is at most $\mathcal{O}(|V_{R,t}|^3)$ added variables. The number of the added constraints is bounded by $\mathcal{O}(|V_{R,t}|^3 + |E|)$ linear inequalities.

3.3 Experiments

We do not have experimental results for our exact formulations and heuristics for CRS computation defined in this chapter. However, we have experimented upper-bound of CRS in [TT00]. These loops are the same experimented in this report and are extracted from various benchmarks (livermore, whetstone, lin-ddot, spec95,..). In that previous work[TT00], we used an old method consisting in unrolling the loop with a certain factor. We define a validity condition for this factor so that the acyclic RS of the unrolled loop body is an upper bound of cyclic RS. In other words, we use loop unrolling to compute the RS of its new body so that it constitutes an upper-bound for cyclic RS. Here is the synthesis of our CRS upper-bound results :

- all the CRS do not exceed 64, that is any SWP schedule will not require more than 64 fp registers ;
- 80.76% of the loops have a CRS ≤ 32 ;
- 76.92% of the loops have a CRS ≤ 16 ;
- 53.84 % of the loops have a CRS ≤ 8 ;

- 34.61 % of the loops have a CRS ≤ 4.

Hence, many loops of our panel do not need adding register constraints during modulo scheduling.

Also, we use in [TT00] an old method for CRS reduction. It consists in adding serial arcs in the unrolled loop and then re-roll it. We have experimentally found that this old method is inefficient (too aggressive). This is why we present a new method in this chapter. Unfortunately, we have no experiments for the moment. However, the old method succeeds in reducing CRS of all loops under 32 fp register while critical circuits increase in 3 cases (6 loops among 27 has a CRS greater than 32). These results show that there are great opportunities for CRS reduction under critical circuit constraints. We are almost sure that our methods described in this chapter would be efficient, even if we use heuristics for solving intLP models.

3.4 Conclusion

This chapter extends RS analysis to innermost loops intended for SWP. Computing CRS is NP-complete and we provide an exact formulation with reduced constraints matrix size. Our heuristics tries to approximate a saturating schedule by decomposing this problem into two steps. First, we compute column numbers so that we maximize the number of values traversing the kernel. In the second step, we build the DAG of the motif and we use our acyclic saturating technique as described in [TT00, Tou01e].

If CRS exceeds the number of available registers, we must reduce it by adding serial arcs into the DDG without increasing the critical circuit if possible. We provide an exact formulation for this NP-hard problem and we prove that it can be reduced to scheduling under register constraints under a fixed execution rate h. If we assume writing offsets, some optimal solutions require, in some cases, to insert null circuits with negative latencies in the extended DDG. These circuits may prevent from finding a software pipelined schedule in the presence of resource constraints. A sufficient an necessary condition to overcome this problem is to guarantee the existence of a topological sort for the retimed loop body. This is done by adding new constraints to the intLP formulation.

Although we do not provide experimental results for our methods described in this chapter, previous experiments in [TT00] have shown that CRS is below 64 in our 27 loops. In many cases, register constraints become redundant and can be evicted from the scheduling process. Also, previous work has shown that there are great opportunities for CRS reduction.

Next chapter extends the notion of register sufficiency to loops. We give our methods to compute it in order to check if spilling is necessary. If spilling isn't avoidable, we give a method to insert load/store operations directly into the DDG to reduce the sufficiency. x

Chapter 4

Cyclic Register Sufficiency

This chapter summarizes our previous work [TE02]. It consists in computing the exact lower bound of register pressure for any SWP schedule. If not enough registers exist, spill code must be introduced into the DDG prior to scheduling. We present our approach, which is in a sense the dual method of [DET00].

This chapter is organized as follows. Section 4.1 defines and studies the concept of cyclic register sufficiency (CRF). We provide an exact method based on integer programming. We also propose a pure algorithmic approximation that decomposes the problem into two parts. The first part is polynomial and is solved via retiming (loop shifting). The second part is NP-complete and is solved with an interval serialization heuristics. Contrary to cyclic register saturation (CRS), the notion of CRF is well studied in the literature. However, most of existing studies focus on fixed initiation intervals. Our work aims to extend this notion to arbitrary execution rates. If CRF exceed the number of available registers, we propose a method that inserts memory operations in Section 4.2, directly into the DDG. This method is a first proposal and is candidate for improvement. Before concluding with a discussion, Section 4.4 shows our experiments.

4.1 Computing Cyclic Register Sufficiency

The cyclic register sufficiency is simply the minimum number of registers required to build at least one valid cyclic schedule:

$$CRF_t(G) = \min_{\sigma \in \Sigma(G)} CRN_t^{\sigma}(G)$$

Contrary to the cyclic register saturation, CRF always exists. This is because the cyclic register requirement is strictly positive, and hence there always exists a schedule which requires $CRF_t(G)$ registers.

The register sufficiency allows us for instance to determine if spill code cannot be avoided for a given loop: if $\mathcal{R}_t$ is the number of available registers of type t, and if $CRF_t(G) \geq \mathcal{R}_t$ then there are not enough registers to schedule the loop. Spill code has to be introduced. Computing CRF is a classical NP-complete problem [EGS95]. Let us begin with an exact formulation.

4.1.1 Exact Formulation

The *absolute* CRF is defined for $\Sigma(G)$, the set of all valid SWP schedule of a DDG. However, In order to be able to use our integer programming formulation in Section 2.3.1 on page 20,

the domain set of our integer variables must be bounded. So, we compute the CRF of a subset $\Sigma_L(G) \subseteq \Sigma(G)$. That is, we compute the minimal register requirement for all valid software pipelined schedules with the property that the total schedule time of one iteration does not exceed L. If L is sufficiently large, then the computed CRF of $\Sigma_L(G)$ is equal to the absolute CRF[1]. Indeed, since the absolute CRF exists necessarily, then there exists a SWP schedule σ that requires $CRF_t(G)$ registers. Hence, its L, the total schedule time of one original iteration, exists and is finite.

Our intLP system answers the question: *"Given a loop $G = (V, E, \delta, \lambda)$, does there exist a software pipelined schedule $\sigma \in \Sigma_L(G)$ of initiation interval h that does not require more than $\mathcal{R}_t$ registers of type t ?"*. We use exactly the same variables and constraints defined for the exact formulation of the cyclic register need in Section 2.3.1 on page 20. We only bound the register requirement of type t:

$$\sum_{\text{acyclic in\_fraction\_of\_}h \text{ interval } I} x_I + \sum_{u^t \in V_{R,t}} p_{u^t} \leq \mathcal{R}_t$$

Solving this intLP system yields to two cases.

1. If a feasible solution is found, then it exists a software pipelined schedule σ such that $CRN_t^\sigma(G) \leq \mathcal{R}_t$. To get the minimal register requirement over all SWP schedules, we must continue by decrementing the upper-bound $\mathcal{R}_t$ (in dichotomic way between $R_{min} = 1$ and $R_{max} = \mathcal{R}_t$) until no solution exists.

2. If no solution exists, then a software pipelined schedule of initiation h such that $CRN_t^\sigma(G) \leq \mathcal{R}_t$ does not exist. We have to increment $\mathcal{R}_t$ (in dichotomic way between $R_{min} = \mathcal{R}_t$ and $R_{max} = CRS_t(G)$) until reaching a solution.

In order to compute CRF, we iterate h starting from h_0 to $h_{max} = L$, i.e. we instantiate an integer programming model for each $h \in [h_0, L]$. Cyclic register sufficiency is the minimum register requirement within all initiation intervals. This method may involve solving too many models. However, the following corollary states that it is sufficient to compute CRF by only instantiating one model with $h = L$ if the upper-bound L is relaxed. Let us start by the following lemma.

Lemma 4.1 *Let $G = (V, E, \delta, \lambda)$ be a DDG of a loop. The minimal register requirement of all the software pipelined schedules $\sigma([rn], [cn], h)$ with an initiation interval $h_0 \leq h \leq L$ is greater or equal to the minimal register requirement of all the software pipelined schedules with an initiation interval $h' = h + 1$ with $L' = L + 1 + \lfloor L/h \rfloor$. Formally:*

$$\min_{\substack{\sigma([rn],[cn],h) \in \Sigma_L(G) \\ h_0 \leq h \leq L}} CRN_t^\sigma(G) \geq \min_{\sigma'([rn'],[cn'],h+1) \in \Sigma_{L+1+\lfloor L/h \rfloor}(G)} CRN_t^{\sigma'}(G)$$

Proof:

[1]We can for instance choose $L = |V| \times \sum_{u \in V} lat(u)$

It is a direct consequence of Proposition 2.1 on page 17. If we increment h by one, we have to increment L by $1 + \lfloor L/h \rfloor$ to get at least one valid software pipelined schedule with the same register requirement :

$$\forall \sigma([rn],[cn],h) \in \Sigma_L(G)/h \leq L, \quad \exists \sigma'([rn'],[cn'],h+1) \in \Sigma_{L+1+\lfloor L/h \rfloor}(G) \ :$$

$$CRN_t^{\sigma'}(G) \leq CRN_t^{\sigma}(G)$$

⌟

Corollary 4.1 *Let $G = (V, E, \delta, \lambda)$ be a DDG of a loop. Then, the exact CRF of G assuming L as an upper-bound of the total schedule time of one iteration is greater or equal to the minimal register requirement with $h = L$ if we relax the upper-bound $L' \geq L$. Formally : :*

$$\min_{\sigma([rn],[cn],h_0 \leq h \leq L) \in \Sigma_L(G)} CRN_t^{\sigma}(G) \geq \min_{\sigma([rn],[cn],L) \in \Sigma_{L'}(G)} CRN_t^{\sigma}(G)$$

where L' is the $(L - h_0)^{th}$ term of the following recurrent sequence ($L' = U_L$) :

$$\begin{cases} U_{h_0} &= L \\ U_{h+1} &= U_h + 1 + \lfloor U_h/h \rfloor \end{cases}$$

Proof :

It is a direct consequence of Lemma 4.1 :

$$\min_{\sigma([rn],[cn],h \leq L) \in \Sigma_L(G)} CRN_t^{\sigma}(G) \geq \min_{\sigma([rn],[cn],h+1) \in \Sigma_{L+1+\lfloor L/h \rfloor}(G)} CRN_t^{\sigma}(G) \geq ...$$

$$... \geq \min_{\sigma([rn],[cn],L) \in \Sigma_{U_{L-h} = U_{L-h-1}+1+\lfloor U_{L-h-1}/(L-h-1) \rfloor}(G)} CRN_t^{\sigma}(G)$$

That is, we relax the upper-bound L at each step, from h to L, i.e. $(L - h)$ times. Since CRF is defined for all initiation intervals, starting from $h = h_0$ amounts to relax the upper-bound $(L - h_0)$ times, as follows.

$$\min_{\sigma([rn],[cn],h_0) \in \Sigma_{L = U_{h_0}}(G)} CRN_t^{\sigma}(G) \geq \min_{\sigma([rn],[cn],h_0+1) \in \Sigma_{U_{h_0+1} = L+1+\lfloor L/h \rfloor}(G)} CRN_t^{\sigma}(G) \geq ...$$

$$... \geq \min_{\sigma([rn],[cn],L) \in \Sigma_{U_L = U_{L-1}+1+\lfloor U_{L-1}/(L-1) \rfloor}(G)} CRN_t^{\sigma}(G)$$

⌟

This corollary enables us to only solve intLP systems with $h = L$; the upper-bound L' must be relaxed. If L is sufficiently large, the computed CRF with $h = L$ is equal to the absolute CRF. Otherwise, we compute a tight lower-bound for CRF. Figure 4.1 draws theoretical asymptotic curves to explain the meanings of Corollary 4.1. If we fix L as an upper-bound of the total schedule time of one iteration, the minimal register requirement under a fixed execution rate h may not be a decreasing function of h. At a certain value of h, the minimal register

requirement may increase if the upper-bound is not relaxed. This behavior has been observed in some of our experiments.

We must be aware that when we combine all register types, a sufficient schedule for all types may not exist. In other words, a software pipelined schedule that needs the exact register sufficiency of *all* types together may not exist. This is because minimizing the register requirement of one type may increase the register requirement of another type. So, some spill operations may be unavoidable even if the register sufficiency of each type is less than the number of available registers. Thereby, we have to bound the register requirement of all types, even if we compute the register sufficiency of only one register type :

$$\forall t \in \mathcal{T} \ : \quad \sum_{u^t \in V_{R,t}} x_{u^t} \leq \mathcal{R}_t$$

These constraints guarantee the existence of at least one schedule that does not require more registers of any type than available.

Our architecture model does not assume any static ILP degree for the target processor : we assume unbounded fine grain parallelism for the considered DDG. Therefore, we can build a kernel as "wide" as possible. This assumption may lead to an under-estimating of the real sufficiency if we target a code with a limited static ILP (superscalar for instance). This is because we cannot always statically specify the parallelism between operations. Thus, some intervals cannot be expressed as parallel[2]. In other words, we cannot ensure that we can always generate a code needing the computed register sufficiency because we cannot statically specify an unlimited instruction parallelism. We propose two choices. First, we continue to assume an unlimited static ILP and leave to the register allocator (to be executed later on) the task of introducing spill code, even if this step of compilation asserts that it isn't necessary. In a second choice, we introduce an upper-bound for the maximal static ILP degree in the model, as follows.

[2]For instance, we cannot specify the instruction $R2 \leftarrow R1 \ \| \ R1 \leftarrow R2$ in superscalar codes.

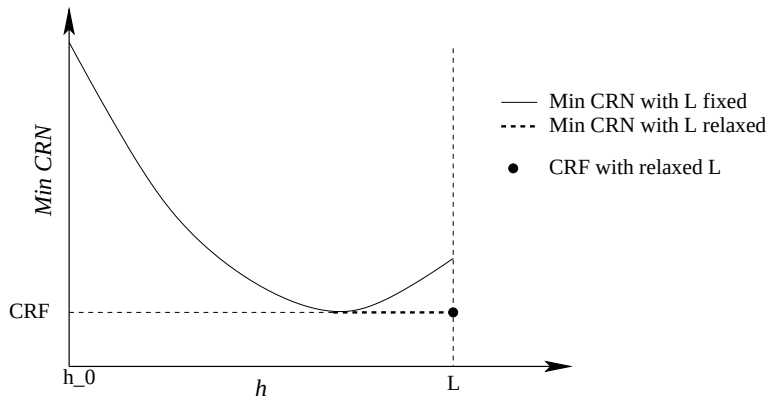


Figure 4.1: Minimal Cyclic Register Need vs. Initiation Intervals

Static Issue Constraints of the Targeted Code As explained above, we must ensure that we can generate a code which needs the computed sufficiency. Let assume that the maximal static issue is $MAXISSUE$. Then, we have to guarantee that we do not schedule more than $MAXISSUE$ operations in parallel. We can model these constraints as previously studied in [Tou01c, Tou01a].

In a modulo scheduled loop $G = (V, E, \delta, \lambda)$, the statements scheduled in parallel in the kernel are those which have the same row number. Our idea is to consider an undirected conflict graph $F = (V, \mathcal{E}_F)$, such a statement u is connected to a statement v iff they are scheduled in parallel. Any clique in this graph represents a set of parallel statements. Then, we have to ensure that the cardinality of any clique in F does not exceeds $MAXISSUE$.

The row numbers of the loop statements are added into the intLP model of the register sufficiency : rn_u is the row number of the statement u. We add in the model the following constraints to compute $rn(u) = \sigma_u \bmod h$:

- introduce the integer variables cn_u for each $u \in V$ which contain the integer division of σ_u by h ;
- write : $\forall u \in V$

$$\begin{cases} \sigma_u = cn_u \times h + rn_u \\ rn_u < h \end{cases}$$

We add in the intLP model a set of binary variables $q_{u,v} \in \{0, 1\}$ for each pair $(u, v)/u \neq v$ set to 1 iff $rn(u) = rn(v)$. There is $(|V| \times (|V| - 1))/2$ introduced $q_{u,v}$ binary variables. We add in the model the following constraints :

- write in the model the linear constraints of

$$q_{u,v} \geq 1 \Longleftrightarrow rn_u = rn_v$$

  There is at most $\mathcal{O}(|V|^2)$ such constraints ;

To simplify the modeling of the cliques in the conflicting graph F, we prefer considering its complementary $F' = (V, \mathcal{E}_{F'})$ such that $(u, v) \in \mathcal{E}_{F'}$ iff u and v are *not* in parallel (i.e. $q_{u,v} = 0$). Hence, any clique in F becomes an independent set if F'. So we add in the model the following :

- introduce a binary variable $y_u \in \{0, 1\}$ for each $u \in V$ set to 1 iff u belongs to an independent set ;
- the independent sets are described by the fact that if two nodes are connected then one of the two does not belong to an independent set. We write in the model :

$$\forall \text{ a couple } (u, v)/u \neq v, \quad q_{u,v} \leq 0 \Longrightarrow y_u + y_v \leq 1$$

  There is at most $\mathcal{O}(|V|^2)$ such constraints ;
- We must not schedule more than $MAXISSUE$ statements in the parallel. This means that the cardinality of any independent set in F' does not exceed $MAXISSUE$:

$$\sum_{u \in V} q_u \leq MAXISSUE$$

We must add at most $\mathcal{O}(|V|^2)$ variables and constraints to specify the fact that no more than $MAXISSUE$ operations are scheduled in parallel. We do not advice this method because it breaks the genericity of the model since it introduces resource constraints.

The next section gives a pure algorithmic heuristics which approximates CRF while overcoming the above problem.

4.1.2 A FCLR Algorithmic Approximation

In this section, we present a First-Columns-Last-Rows (FCLR) heuristics which constructs a software pipelining motif for approximating the register sufficiency of a register type. Our heuristics is the minimization version of the one explained in the previous Section 3.1.2. It consists of the following steps.

1. We first find column numbers that minimize the number of traversed iterations by a value. This intends to minimize the total number of values copies (turns around the circle). We can change the objective function of the intLP system in Section 3.1.2 from maximization into minimization. This problem has been solved by Leiserson and Saxe with an optimal algorithm via retiming with a polynomial complexity in [LS91]. We explain their method below.

2. Once the column numbers computed, we build the DAG with respect to the inner-motif dependences and the entry/exit values as detailed in Section 3.1.2. We must minimize the interference between the lifetime intervals inside the motif, i.e. within the DAG just built. We use the DAG technique of the register sufficiency studied in [TT00] (interval serialization) to construct an acyclic schedule $\underline{\sigma}$ which requires a minimized number of registers. We are sure that we can construct such an acyclic schedule with any static ILP, and hence any SWP scheduler can build a kernel that satisfies any static ILP constrained by the underlying processor. Row numbers are set to:

$$\forall u \in V \ : \qquad rn(u) = \underline{\sigma}(u)$$

3. We choose a valid initiation interval with respect to the inter-kernel dependences and the number of traversed motifs:

$$h = \max_{u \in V} rn(u) + lat(u)$$

4. Since we have computed $\sigma(h, [rn], [cn])$, a software pipelined schedule $\sigma \in \Sigma_L(G)$ minimizing the register need is completely defined. The approximated register sufficiency is :

$$CRF_t^*(G) = CRN_t^{\sigma}(G)$$

Steps 2, 3, and 4 have been detailed in Section 3.1.2. Step 1 is different since it can be performed with polynomial optimal algorithms. Next section gives more details about column numbers computation.

Computing Column Numbers with Retiming Originally, retiming was intended for synchronous circuit design. In this area, a register has a different meaning, let us call it *circuit register*. A distance in each flow arc represents the number of circuit registers needed to pass the computed values. That is, if there are two flow arcs $e_1 = (u, v)$ and $e_2 = (u, v')$ coming from the same node but going to two distinct consumers, the number of required circuit registers , in the field of circuit design, is $\lambda(e_1) + \lambda(e_2)$: there is no sharing between the two flows. Leiserson and Saxe proved that seeking a retiming with a minimal number of circuit registers can be reduced to minimum cost flow [LS91], a well solved polynomial problem with lots of optimal algorithms [EK72, GT86, Orl88]. They assume identical registers, i.e. all nodes represent values and all arcs are flows. We show at the last how to consider different types of registers.

Problem 4.1 (Minimum Cost Flow Problem) *Let $G = (V, E)$ be a directed graph. For each arc $e \in E$, we call $cap(e) \in \mathbb{R}^+$ the capacity of e. A flow is a function f which associates with each arc a positive real $f(e) \in \mathbb{R}^+$ with the following properties :*

$$\begin{array}{ll} \forall e \in E & 0 \leq f(e) \leq cap(e) \\ \forall u \in V & \sum_{?\xrightarrow{e}u} f(e) = \sum_{u\xrightarrow{e}?} f(e) \end{array}$$

A cost function associates with each arc e a cost $\omega(e)$. The minimum cost problem is to find a flow f for G which minimizes

$$\sum_{e \in E} f(e)\omega(e)$$

A variant of the min-cost flow problem, also polynomial, adds supply/demand parameters. The flow has to guarantee :

$$\forall u \in V \quad \sum_{?\xrightarrow{e}u} f(e) - \sum_{u\xrightarrow{e}?} f(e) = b_u$$

where $b_u \in \mathbb{N}$ is the supply/demand parameter of the node u. As explained in Chapter 2, retiming a loop consists in computing a shift $r(u)$ for each statement u which delays the operation $u(i)$ with $r(u)$ iterations. Each original distance $\lambda(e)$ of an arc $e = (u, v)$ becomes $\lambda_r(e) = \lambda(e) - r(u) + r(v)$ in the retimed loop.

Problem 4.2 (State-Minimization Problem [LS91]) *Let $G = (V, E, \delta, \lambda)$ be a circuit. The state-minimization problem is to find a valid retiming such that $S(G_r)$ the total state of the retimed loop is minimized, in which*

$$S(G_r) = \sum_{e \in E} \lambda_r(e)$$

$S(G_r)$ can be rewritten as :

$$S(G_r) = \sum_{e=(u,v)\in E} \lambda(e) + r(v) - r(u) = S(G) + \sum_{u \in V} r(u).(d_G^-(u) - d_G^+(u))$$

$S(G)$ is constant (sum of all dependence distances), hence minimizing $S_r(G)$ is equivalent to minimizing

$$\sum_{u \in V} r(u).(d_G^-(u) - d_G^+(u)) \tag{4.1}$$

which is a linear function of $r(u)$ since the indegree and outdegree are constant for u. This minimization is constrained by the retiming validity, i.e. the fact that all register counts $\lambda_r(e) = \lambda(e) + r(v) - r(u)$ are positive:

$$\forall e = (u, v) \in E : \quad \lambda(e) - r(v) + r(u) \geq 0 \tag{4.2}$$

Leiserson and Saxe [LS91] showed that the intLP defined by (4.1) and (4.2) can be recast into a min-cost flow by considering the dual problem of this intLP. Then, we look for a flow $f(e)$ for each arc such that:

$$\sum_{u \xrightarrow{e} ?} f(e) - \sum_{? \xrightarrow{e} u} f(e) = d_G^-(u) - d_G^+(u) \tag{4.3}$$

while the total cost $\sum_{e \in E} f(e)\lambda(e)$ is minimized. Each arc has a cost $\lambda(e)$ with infinite capacity. After computing the optimal minimum cost flow, the shifts $r(u)$ are the dual variables (potentials) of the optimal flow f^*, computed by most existing algorithms.

Another variant of the state minimization problem, that also can be reduced to minimum cost flow, includes $\beta(e)$, a real cost to each arc called a *breadth*. This breadth models some special constraints to circuit design where adding circuit registers has different costs depending on flow arcs. Then, the state minimization problem minimizes

$$\sum_{u \in V} r(u)(\sum_{? \xrightarrow{e} u} \beta(e) - \sum_{u \xrightarrow{e} ?} \beta(e))$$

However, the state minimization problem does not exactly compute our column numbers. This is because circuit registers are not shared, and each arc e uses $\lambda_r(e)$ circuit registers. In our case, we need to minimize the total numbers of traversed motifs, i.e. to minimize

$$\sum_{u \in V} \max_{u \xrightarrow{e} ? \in E} \lambda_r(e)$$

In terms of circuit design, it means that we wish to share the largest possible number of circuit registers between different arcs (with greatest register counts). Leiserson and Saxe give a solution for this problem by using a trick. Figure 4.2 is an example. Part (a) shows a DDG in which a statement u writes a value read by k consumers. If we use state minimization algorithm on this DDG as it is, it considers that the value coming from the statement u and going to the k consumers needs distinct registers. This is not true since a value read by more than one consumer resides in only one register. So, we must model sharing. It means that a value resides in a register until the last iteration needed. The result of the retiming must give a minimal register count $S(G_r) = \sum_{u \in V}(\max_{e=(u,v)} \lambda_r(e))$. This is done by transforming the DDG as follows (see Part (b)):

first we assume a breadth (cost) equal to $\beta(e) = 1/k$ for each arc;

second we add a virtual node $\hat{u}$;

finally we connect each consumer v_i to $\hat{u}$ by an arc $\hat{e}_i$ with breadth $\beta(\hat{e}_i) = 1/k$ and distance $\lambda(\hat{e}_i) = \lambda_{max} - \lambda(e_i)$, with $\lambda_{max} = \max_{1 \leq i \leq k} \lambda(e_i)$. Then, all paths from u to $\hat{u}$ have the same distance ($= \lambda_{max}$).

Now, we are ready to re-time this DDG by minimum cost flow algorithms as mentioned before. It is easy to see that retiming this transformed graph gives the expected result.

1. Since the dummy node $\hat{u}$ is a sink of the graph, retiming this graph makes the distances $\lambda_r(\hat{e}_i)$ as small as possible because they are not constrained by any circuit (the dummy node $\hat{u}$ is a sink). Then, one of these virtual arcs $\hat{e}_j$ $(1 \leq j \leq k)$ gets a null retimed distance $\lambda_r(\hat{e}_j) = 0$.

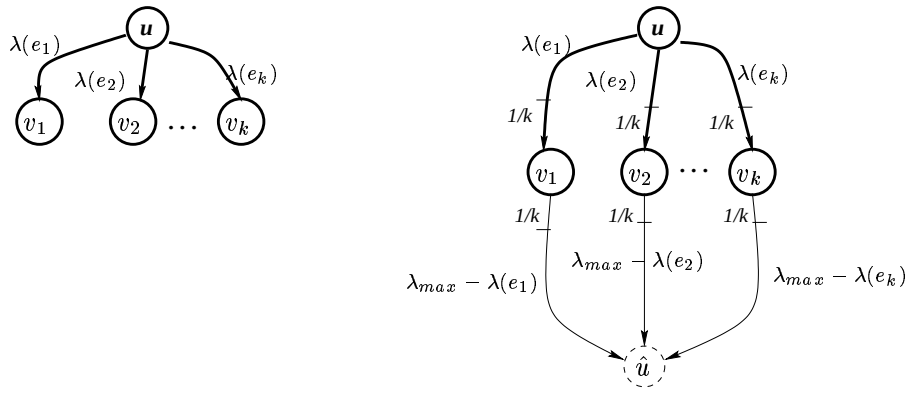
2. Retiming has the property of preserving the sum of distances of the paths $u \rightsquigarrow \hat{u}$, that is $\lambda_r(u \rightsquigarrow \hat{u})$ is the same for all the paths from u to $\hat{u}$ since they are identical in the un-retimed circuit $(= \lambda_{max})$. By considering the path $u \rightarrow \hat{e}_j \rightarrow \hat{u}$ where $\lambda_r(\hat{e}_j) = 0$, its distance is $\max_{0 \leq j \leq k} \lambda_r(e_i)$ which is the distance of every path from u to $\hat{u}$. Sharing is completely defined.

3. Since each arc has a breadth $1/k$, the total register count is

$$\sum_{1 \leq j \leq k} 1/k \max_{1 \leq j \leq k} \lambda_r(e_i) = \max_{1 \leq j \leq k} \lambda_r(e_i)$$

Retiming the transformed DDG gives us column numbers $cn(u) = r(u)$, where the number of traversed motifs is minimized. Now, in the presence of multiple register types, some arcs do not represent flows if we consider one of the types. To handle these serial arcs, we only have to set their breadth to 0 to model the fact that they do not require any register of the type we consider. Accordingly, the register count (objective function) computes only values of the desired type. Our algorithmic approximation of CRF is now completely defined.

Next section investigates spill code insertion if CRF is higher than the number of available registers.



(a) Original DDG

(b) Transformed DDG to be Retimed

Figure 4.2: Retiming DDGs with Maximal Register Sharing

4.2 Reducing Cyclic Register Sufficiency

If the register sufficiency of a loop $G = (V, E, \delta, \lambda)$ is $CRF_t(G) > \mathcal{R}_t$, we have not enough registers to pursue the computation and hence spill code must be introduced. In this section, we show how storing some variables in memory decreases the sufficiency, assuming a RISC processor (load-store architecture).

Adding extra memory operations may cause cache misses which dramatically decrease the performances, especially in VLIW architectures where long memory access delays are not dynamically recovered as in superscalar processors. Since memory access latencies are hardly statically foreseeable, we try to minimize the amount of inserted load/store operations[3]. Furthermore, adding them directly into the DDG before scheduling is better than after scheduling. This is because we cannot guarantee the existence of free slots for additional load/store operations in a scheduled code. This leads to an iterative spilling and rescheduling. The method discussed in this section is a first approach and is candidate to improvement : it gives full priority to spilling those values passed to farthest iterations.

If $CRF_t(G) > \mathcal{R}_t$, at least $S = CRF_t(G) - \mathcal{R}_t$ values of type t have to be spilled. Our heuristics proceeds by preventing some values from being alive during successive iterations by storing them in memory. However, since CRF is likely constrained by dependence circuits, we must privilege the values that belong to a circuit in the DDG[4]. This is because, while loop retiming does not change the sum of distances in a circuit, the values that do not belong to a circuit can be shifted via loop retiming so as to belong to the loop body. Consequently, the number of iterations that they span is reduced. Our aim is to first reduce circuit distances. The skeleton of our method is described as follows.

1. Build $FarCons$ a sorted list of the values depending on the number of iterations they span. Each value u^t may span $\max_{e=(u,v)\in E_{R,t}} \lambda(e)$ iterations. We first give the priority to values that belong to a circuit. Then, we sort them in decreasing order of distances. For instance, the value u in Figure 4.3.(a) may span $max(\lambda_1, \lambda_2)$ iterations. If no inter-iterations value exists (i.e. all the flow distances are null), go to 4.

2. Pick up the first value in the list which crosses $\lambda > 0$ iterations. We may potentially reduce the register sufficiency by λ registers if we prevent the value from being alive in a register after exiting an iteration. We spill this value by considering one of the two following code transformations.

   (a) Store the value at the iteration where it is defined, and load it for each consumer as described in Figure 4.3.(b). This transformation may reduce the sufficiency by $\max_{e=(u,v)\in E_{R,t}} \lambda(e)$ registers but it inserts $|Cons(u^t)|$ loads.

   (b) Store the value at the iteration where it is defined, and load it for only the first consumer in terms of dependence distance as described in Figure 4.3.(c)[5]. This transformation inserts only one load but may reduce the sufficiency by only $\min_{e=(u,v)\in E_{R,t}} \lambda(e)$ registers.

[3] Inserting minimal spill operations is a classical NP-complete problem [BS76, Car91, FL98].

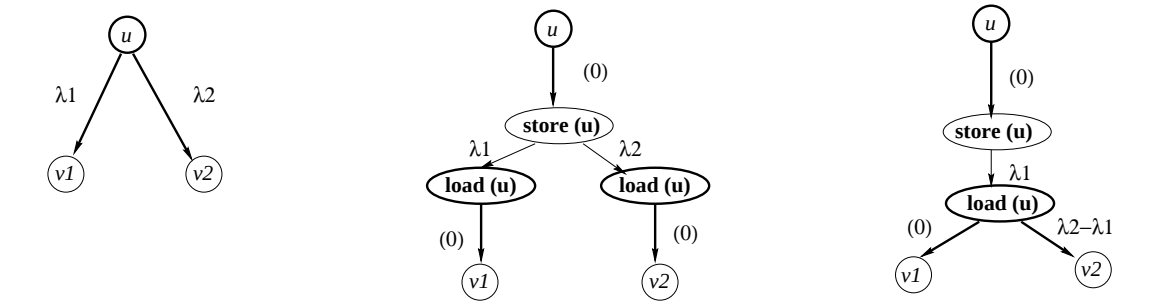
[4] They are determined by looking if there exists a path from each statement to itself in the DDG. ALL_PAIR_SHORTEST_PATH, for instance, can be applied.

[5] In this example, we suppose that $\lambda_1 \leq \lambda_2$.

If S is smaller than the number of spanned iterations by this value, we do not need to store the value in the memory during all the iterations it spans. In general, we spill the values until the number of "saved" iterations is S.

3. If $CRF_t(G) > \mathcal{R}_t$, go to 1, else exit.

4. At this point, the cyclic register sufficiency is still greater than $\mathcal{R}_t$ while all the values are consumed in the same loop body (i.e. all the flow distances are null). Then, we reduce the acyclic register sufficiency of the loop body only (described in Section 4.3).



(a) Inter-iterations values

(b) Spilling for eliminating the values traversing the iterations

(c) Spilling for reducing the number of values traversing the iterations

Figure 4.3: Register Spilling in Loops

A possible limitation of this approach arises if a value u belongs to a circuit but has a flow arc e that does not belong to any circuit (see Figure 4.4). If this flow arc has the largest distance among other flow arcs exiting from u, it will be chosen for spilling (see (u, v) in Figure 4.4). This may not be a good choice, since the distance of this arc can be reduced by retiming. To overcome this limitation, we must first retime the graph, before spilling, so as to reduce the distances of arcs that do not belong to a circuit. For this purpose, we use the Leiserson and Saxe method that minimizes the register count with maximal register sharing, as described in Section 4.1.2.

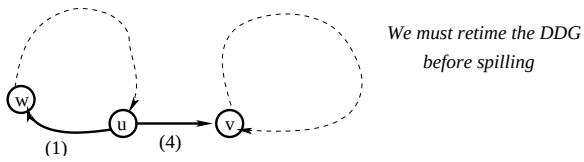
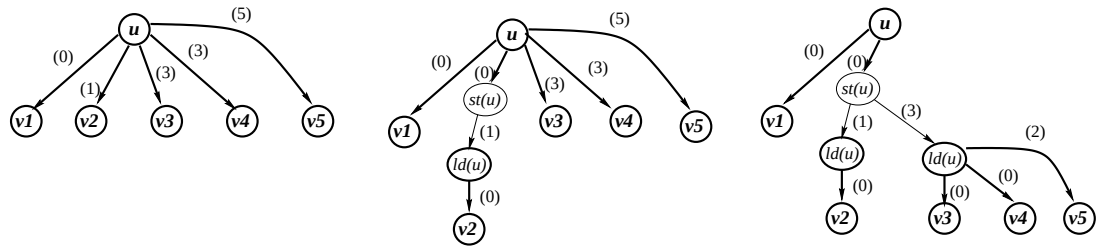


Figure 4.4: Values in Circuits

The steps of our heuristics are detailed in Algorithm 1. Our heuristics creates an *active* list which contains the values that are alive during successive iterations. If $S = CRF_t(G) - \mathcal{R}_t > 0$,

we have to prevent some values in *active* from being alive for at least S successive iterations. If a value is produced at the current iteration and consumed λ iterations later, we have to store it in the current iteration and load it λ iterations later hoping that we reduce the cyclic sufficiency by λ. This creates a new dependence between the store and the load with distance λ, but this dependence is through memory and hence does not consume any register. Since we have to save at least S registers, we save inter-iteration values in memory until the sum of the introduced dependence distances becomes at least S. The *active* list is sorted by decreasing distances to give the highest priority to the value which traverses the maximum number of iterations. If a value is consumed by more than one operation in the successive iterations $(\lambda_1, \cdots, \lambda_n)$, we load this value for every further consumer[6] until we reach S saved iterations, i.e. when we load it for the consumer of the $\lambda_k \geq S$ iteration later. Our algorithm optimizes the number of inserted loads where it reaches at least S saved iterations by connecting the last inserted load (the load of the λ_k^{th} iteration) to the remaining consumers $(\lambda_{k+1}, \cdots, \lambda_n)$, as shown in Figure 4.3.(c).



(a) u traverses 5 iterations (b) spill for 1 iteration (c) spill for 3 iterations

Figure 4.5: Spilling for Reducing Cyclic Sufficiency

Example 4.2.1 *Let us give an example to understand how Algorithm 1 works. Figure 4.5.(a) is a part of a DDG (arcs are labeled by distances) where we need to reduce its cyclic sufficiency by $S = 3$ registers. The value u traverses 5 iterations so we would spill it in successive steps, as follows.*

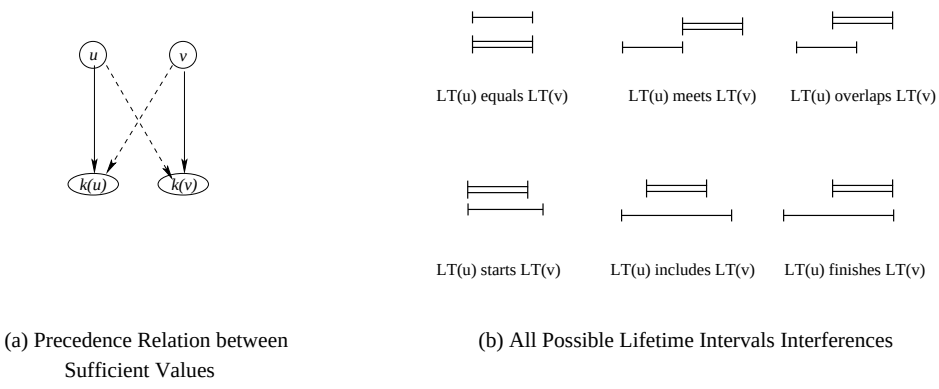
1. *At the beginning, active $= \{u\}$ and $FarCons = \{v_2, v_3, v_4, v_5\}$ are sorted by increasing distance ;*
2. *We first begin by storing u.*
3. *The first consumer in $FarCons$ is v_2, so we load u $\lambda_{v_2} = 1$ iteration later as shown in Figure 4.5.(b). The number of saved iterations is $1 < S$, so we need to perform loading u for the remaining consumers.*
4. *At this step, $FarCons = \{v_3, v_4, v_5\}$. We load u for $\lambda_{v_3} = 3$ iterations later as shown in Figure 4.5.(c). The number of saved iterations is $3 = S$. We have saved enough iterations, and the remaining consumers $FarCons = \{v_4, v_5\}$ use the last load.*
5. *Repeat this step until the cyclic register sufficiency is below the target limit. In the case of unsuccess, i.e. when active is empty, transforming the loop-carried dependences from registers into loop-carried dependences through memory is not sufficient. We must decrease the register sufficiency in the DAG of loop body itself as explained in Section 4.3.*

[6]except those that belong the current iteration since they need the value produced at the same iteration

4.3 Reducing Acyclic Register Sufficiency

If RF is greater than $\mathcal{R}_t$, then we introduce spill code in the DAG to reduce its sufficiency. Our strategy relies on minimizing introduced load-store operations.

Before spilling, we must detect which values are *sufficient*, i.e. which ones are always simultaneously alive. We use our value serialization algorithm with a target $\mathcal{R} = 1$ to compute them. The resulted extended DAG has the following properties:



(a) Precedence Relation between Sufficient Values

(b) All Possible Lifetime Intervals Interferences

Figure 4.6: Sufficient Values Property

1. the register saturation of the extended DAG cannot be reduced, and hence it is equal to its register sufficiency;
2. saturating values of the extended DAG are the sufficient ones;
3. any two sufficient values $u, v \in V_{R,t}$ are always simultaneously alive for any schedule. That is, we cannot serialize the lifetime interval of u before the lifetime interval of v, and vice versa. Hence, they must satisfy the following necessary and sufficient condition (see Figure 4.6):

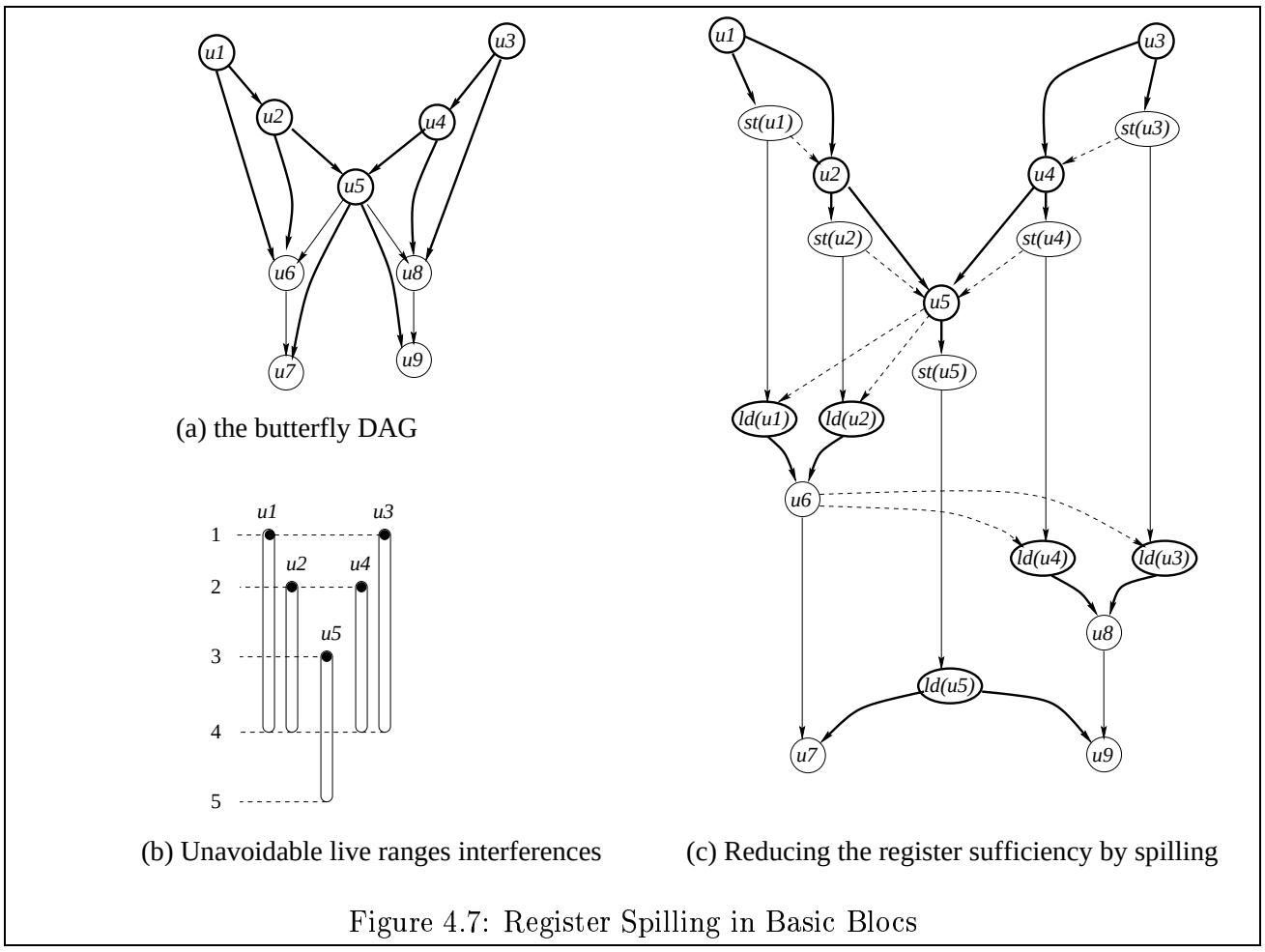
$$\begin{aligned} & v < k(u) \wedge \ u < k(v) \\ & \wedge \ lp\big(u, k(v)\big) > \delta_w(u) - \delta_r\big(k(v)\big) \\ & \wedge \ lp\big(v, k(u)\big) > \delta_w(v) - \delta_r\big(k(u)\big) \end{aligned} \tag{4.4}$$

in which $u < v$ means that it exists a path from u to v, and $lp(u, v)$ denotes the size of the longest path from u to v. $k(u)$ is the killer[7] of u defined by the saturating killing function. This condition is necessary since it prohibits any serialization of the lifetime intervals, otherwise we introduce a circuit. It is sufficient since if two values satisfy this condition, then their lifetime intervals are in conflict necessarily.

Our algorithm iteratively inserts spill code until it reaches the target sufficiency. As an example, Figure 4.7.(a) is a DAG in a butterfly shape such that its RS is 5 (we ignore the

[7]Note that $k(u)$ and $k(v)$ are not necessarily distinct.

latencies for clarity reasons), where values are in bold circles and flow arcs in bold lines. Its RS cannot be decreased and hence its RF is equal to 5 too. We want to reduce the register sufficiency to 2. The sufficient (saturating) values are $\{u_1, u_2, u_3, u_4, u_5\}$ since any pair of them satisfies Condition (4.4).



(a) the butterfly DAG

(b) Unavoidable live ranges interferences

(c) Reducing the register sufficiency by spilling

Figure 4.7: Register Spilling in Basic Blocs

Condition (4.4) defines an intrinsic relative order between lifetime intervals of the sufficient values, as illustrated in Figure 4.7.(b). We use this ordering to guide our spilling algorithm. We name *integer point* the logical time (relative date) when a value is defined or killed. The points are graduated starting from 1 according to the relative order defined by the precedence relation $<$. As an example, (1,2,3,4,5) are five points in Figure 4.7.(b). Note that some definition or kill events are not related by any precedence relation (as the definition of u_1 and u_3) and may be assigned to the same point. We will use these points to highlight the regions (date intervals) where the register need exceeds the desired one. For instance, the excessive regions in Figure 4.7.(b) are $[2, 3]$ because it requires 4 registers, and $[3, 4]$ because it requires 5 registers. Since we define a virtual dating, we assign to each value a *definition point* $dp(u)$ and a *kill point* $kp(u)$. The relative order we define enables to use any efficient spilling strategy in the literature.

The register need changes only at the dating points, i.e. at the beginning or at the end of an interval. The sufficient values are managed in a sorted list in increasing order of definition

points. Thanks to this list, our algorithm can quickly scan forward the live ranges by skipping from one definition point to the next one. Our strategy is inspired from the Poletto approach [PS99] applied to spill code insertion for straight-line code. His heuristics has a linear complexity with good experimental results. It is explained below.

The algorithm iterates over the integer dating points starting from 1. At each step, we maintain an *active* list of live ranges which overlap the current point. The active list is kept sorted in increasing order of end points. For each new life interval, the algorithm scans the active list to remove any expired value, i.e. the one which has been necessarily killed when treating the current dating point. When the length l of the active list is greater than $\mathcal{R}_t$, at least $l - \mathcal{R}_t$ values must be spilled. There are several possible heuristics for selecting which value to spill. We can for instance choose the one that do not increase the critical path. We prefer to minimize the amount of introduced spill code. Our heuristics selects the the value which would be the last killed. Since the active list is sorted, this values is the last item in the active list. For each spilled value u, we insert a store operation. Poletto approach loads the stored value for every use: the spilled life interval is splited into several small parts and the original interval is removed from the active list. This aggressive approach may insert an excessive number of loads.

The algorithm iterates until the register sufficiency is reduced[8] to $\mathcal{R}_t = 2$. Figure 4.7.(c) gives the resulted DAG in which all the values have been spilled because of high register pressure. Dashed arcs represent the serial arcs added for reducing the register saturation to 2 to show that the register sufficiency of this DAG is 2 too. Note that these dashed arcs are not present in the final DAG (they are shown to only prove that the saturation can be reduced to 2).

Now, we give full algorithms for our heuristics explained above. We start by defining the dating points.

4.3.1 Relative Dating

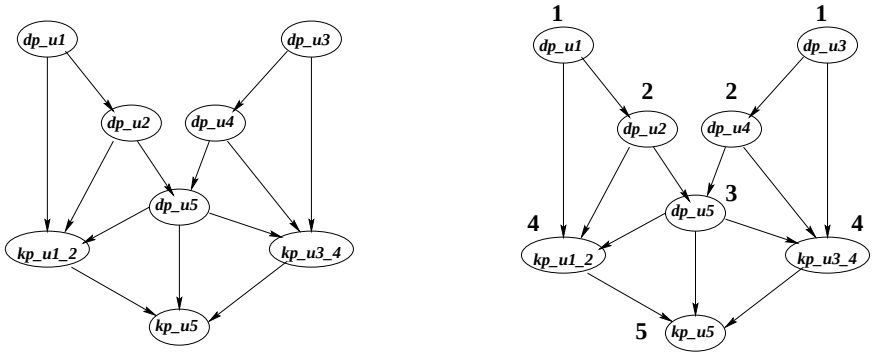
We build a DAG which reflects relative order between value definitions and kills.

Definition 4.1 (Dating DAG) *Let $G = (V, E, \delta)$ be a DAG. A dating DAG, noted $G^t_d = (V_d, E_d)$, and associated with G for register type t is defined by:*

- $V_d = \{dp_u, kp_u \,/\, u \in V_{R,t}\}$. *$dp_u$ corresponds to the definition node of the value u^t. kp_u corresponds to the killing node $k(u)$ of the value u^t defined by the saturating killing function of G. If some values share the same killer, they necessarily share the same kp. Any node in V_d (killing or definition node) is called a dating node;*
- $\forall dp_u, kp_u \in V_d \qquad (dp_u, kp_u) \in E_d$;
- $\forall u, v \in V_{R,t} \qquad (dp_v, kp_u), (dp_u, kp_v) \in E_d \iff u, v$ *satisfy Condition 4.4*
- $\forall u, v \in V_{R,t} \qquad (kp_u, kp_v) \in E_d \iff k(u) < k(v) \ \wedge \ lp\big(k(u), k(v)\big) \geq \delta_{r,t}(u) - \delta_{r,t}(v)$

[8] It is clear that in the presence of a RISC architecture with n-ary statements, we cannot use less than n registers since we need at least n distinct operands to execute the statement. Here, we have binary statements with two distinct operands, so we cannot reduce the sufficiency below 2. We could imagine another architecture where this number is higher $n > 2$, or this number $n = 1$ (unary operations).

A dating DAG must be simplified by transitive reduction, i.e. we remove obsolete transitive arcs. Figure 4.8.(a) is the dating DAG of Figure 4.7.(a) after removing transitive arcs: for instance, the arc (dp_{u1}, dp_{u5}) has been removed because there exists a path $(dp_{u1}, dp_{u2}, dp_{u5})$. Note that the dating nodes dp_{u1} and dp_{u2} share the same killing node kp_{u1_2} because the values u_1 and u_2 have the same saturating killer in G ($k(u_1) = k(u_2) = u_6$).



(a) Dating DAG (b) Dating Function

Figure 4.8: Relative Dating

A *dating function* assigns an integer starting from 1 to each dating node in G_d^t in which $date(d) \leq date(d')$ iff $d < d'$ in G_d^t. A topological sort of G_d^t is a dating function. However, since the dating points are used in our heuristics to sort the life intervals assuming possible parallelism between the operations, a topological sort is not really appropriate because it assigns different dates to two dating nodes even if they are not constrained by any precedence relation. This fact influences the results of the spilling decision because it defines a kind of priority. We prefer to define the dating function by an "as soon as possible" schedule of G_d^t. This enables us to give the same dating point to parallel dating nodes, see Figure 4.8.(b). At this step, each value $u \in V_{R,t}$ has an integer definition point $dp(u)$ and a killing point $kp(u)$ which are defined by the dating function.

4.3.2 Algorithms for Reducing Acyclic Register Sufficiency

Algorithm 2 presents our techniques to reduce the RF. It maintains an active list of current alive values. At each definition point, it spills the last killed values if the register requirement exceeds the target sufficiency. Algorithm 3 removes killed values when it reaches a definition point. Algorithm 4 defines the inserted memory operations and arcs resulting from spilling: note that this algorithm aggressively inserts a load for each read in order to split the original life interval into several small parts. We can optimize it by doing a post-pass for reducing the number of inserted loads by merging two small live ranges if they do not increase the RF (remove those that do not belongs to an excessive region).

4.4 Experiments

Optimal CRF of our loop benchmarks are computed by integer programming. Table 4.1 summarizes our results. We remark that some loops (such as spec-spice-loop4) have a non negligible sufficiency compared to the number of statements because of their intrinsic register pressure (data dependence relations between the statements). Depending on the number of available registers, we may not avoid spilling. Other loops (as spec-dod-loop2) have a low sufficiency compared to the number of statements. We do not have experimental results for our heuristics, nor those about spilling strategy. We discuss both in the next section.

Loop	$\|V\|$	$\|V_R\|$	CRF
Lin-ddot	4	4	3
Liv-loop1	9	8	4
Liv-loop23	20	19	NA
Liv-loop5	5	4	2
Spec-dod-loop1	13	12	NA
Spec-dod-loop2	10	9	2
Spec-dod-loop3	11	10	3
Spec-dod-loop7	4	3	2
Spec-tom-loop1	15	12	NA
Spec-fp-loop1	5	4	2
Spec-spice-loop1	2	2	2
Spec-spice-loop2	9	9	4
Spec-spice-loop3	4	3	2
Spec-spice-loop4	12	8	8
Spec-spice-loop5	2	1	1
Spec-spice-loop6	6	6	4
Spec-spice-loop7	5	4	3
Spec-spice-loop8	4	3	3
Spec-spice-loop9	11	9	4
Spec-spice-loop10	4	3	2
Whet-cycle4_1	4	4	1
Whet-cycle4_2	4	4	2
Whet-cycle4_4	4	4	4
Whet-cycle4_8	4	4	8
Whet-loop1	16	16	5
Whet-loop2	7	6	3
Whet-loop3	5	5	4

Table 4.1: Optimal Cyclic Register Sufficiency

4.5 Discussion and Conclusion

This chapter investigates the cyclic register sufficiency problem which computes the minimal register need for all valid schedules. The exact formulation is based on CRS intLP model but with bounding the register requirement. This is because we express the exact cyclic register need according to an arbitrary SWP kernel.

As in the acyclic case, optimal CRF assumes infinite parallelism. This leads to underestimate the real CRF since the target code has limited static ILP. To overcome this problem, we propose a pure algorithmic approximation that looks for a sufficient SWP by decomposing the problem into two parts. The first part seeks column numbers that minimize the total number of inter-kernel values. This is a polynomial problem solved via retiming by Leiserson and Saxe in [LS91] by using minimum cost flow algorithms. The second part of the problem looks for row numbers that minimize the total number of intra-kernel values simultaneously alive. Since the operations belonging to the kernel have been fixed, it remains to compute an issue slot for them. A DAG is built as in CRS computation by adding entry and exit values with the inter-kernel flows. Accordingly, the problem becomes an acyclic scheduling with limited registers (NP-complete). We use our acyclic RF computation technique which reduces the RS as minimum as possible by setting $\mathcal{R} = 1$. This technique is nearly optimal and guarantees the existence of a kernel with any static ILP degree.

Our proposed spilling strategy is, in a sense, the dual method of [DET00]. We add load/store operations to prevent values from being alive during farthest iterations. We insert spill code directly into the DDG before scheduling phase. Existing techniques perform scheduling before spilling, so they have to recompute the schedule when adding extra load/store operations. This leads to iteratively apply scheduling followed by spilling until finding a solution. Early spilling is a better approach since it reduces CRF and guarantees the existence of a least one valid SWP schedule.

Some studies [LMEG96, Jan01] claim that inserting spill code in modulo scheduled loops is better than increasing the II. We do not adhere to this thesis at all. The reason why these authors make this claim is that, first, they assume static memory operation latencies and they remark after experiments that the SWP scheduler succeeds (in most cases) in finding free slots for inserted spill code. Second, they confuse static loop performance defined by the computed II and real (dynamic) one. Memory access operations have unforeseeable effects and may play havoc with the computed schedule. Of course, we can be optimistic and assume that spilled values reside in cache. We do not prefer such assumption because any misprediction leads to cache misses which result in deep performance loss. On superscalar processors, the fluidity of the dynamic execution of the pipelined loop is broken since long miss latency scrambles the static schedule. Reusing registers is a better choice, since OoO processors can dynamically eliminate anti-dependences with register renaming. Also, a VLIW machine completely stalls because of cache misses. We prefer keeping the dynamic execution under a static control when possible.

Before inserting spill code, we must understand why the sufficiency may be higher than $\mathcal{R}_t$. If the data dependence graph is extracted from an original loop using a high level programming language, the flow dependence are specified implicitly through variables in memory (arrays for

instance). Compilers generally transform the high level code into an intermediate one, where each reference to the memory is replaced by a pair of load/store. At this point, the spill code exists in the loop, and the cyclic register sufficiency is low enough, since all the values are loaded from the memory at each use. Then, compilers make some load/store optimizations [CCK90, DGS93, BG96, DET00] in order to remove redundant memory operations, and to exhibit more parallelism. Flow dependences during this optimization phase are transformed from memory dependences to register ones. Register sufficiency increases as a consequence. Then, we must have a tradeoff between redundant memory elimination process and CRF increase. Instead of eliminating all the redundant load/store, we must care not to increase the sufficiency more than $\mathcal{R}_t$. We think that cyclic register sufficiency must intervene during the load-store optimization process to keep some of the original spill code instead of inserting new one.

Next chapter investigates another approach of handling register pressure. Instead of analyzing CRS and CRF before scheduling, we build a cyclic register allocation directly into the DDG without hurting intrinsic ILP.

Algorithm 1 Reducing the Cyclic Register Sufficiency

Require: A DDG $G = (V, E, \delta, \lambda)$ and a target cyclic sufficiency $\mathcal{R}_t$

retime G with minimal register count, maximal register sharing{Leiserson and Saxe Algorithm}
while $CRF_t(G) > \mathcal{R}_t$ **do**
$S \leftarrow CRF_t(G) - \mathcal{R}_t$ {We must spill for at least S iterations}
$values\_in\_circuits \leftarrow \{u \in V_{R,t} / u \in circuit \wedge \exists e = (u, v) \in E_{R,t} \wedge \lambda(e) > 0\}$ {sorted by decreasing order of distances}
$values\_not\_\_in\_circuits \leftarrow \{u \in V_{R,t} / u \notin circuit \wedge \exists e = (u, v) \in E_{R,t} \wedge \lambda(e) > 0\}$ {sorted by decreasing order of distances}
build *active* by merging the list $values\_in\_circuits$ before the list $values\_not\_\_in\_circuits$
if $active = \{\}$ **then** {no inter-iteration values exist}
reduce the acyclic sufficiency in the loop body DAG (see Section 4.3)
exit
end if
while $S > 0$ **do**
for all $u \in active$ in the priority order **do**
build $FarCons \leftarrow \{v \in Cons(u^t) / e = (u, v) \in E_{R,t} \wedge \lambda(e) = \lambda_v > 0\}$ a list of further consumers in increasing order of distances
insert $store(u)$ in G
insert a flow arc $e = (u, store)$ into G with $\delta(e) = lat(u)$ et $\lambda(e) = 0$
$saved_u \leftarrow 0$ {contains the number of saved iterations for u}
for all $v \in FarCons$ in the priority order **do**
remove v from $FarCons$
insert $l = load(u)$ in G
$LastLoad_u = l$
$\lambda_{last} = \lambda_v$ {the latest iterations when a load occurs}
remove the flow arc (u, v) from G
insert the arc $e = (store(u), l)$ into G with $\delta(e) = lat(st)$ and $\lambda(e) = \lambda_v$
insert the flow arc $e = (l, v)$ into G with $\delta(e) = lat(load)$ and $\lambda(e) = 0$
$saved_u \leftarrow saved_u + \lambda_v$
if $saved_u \geq S$ **then** {we have saved enough iterations}
break
end if
end for
if $saved \geq S$ **then** {use the latest load for the remaining farther consumers}
for all $v \in FarCons$ in the priority order **do**
remove the flow arc (u, v) from G
insert a flow arc $e = (LastLoad_u, v)$ into G with $\delta(e) = lat(st)$ and $\lambda(e) = \lambda_v - \lambda_{last}$
end for
end if
end for
$S \leftarrow S - saved_u$
end while
end while

Algorithm 2 Reducing Acyclic Register Sufficiency

Require: a DAG $G = (V, E, \delta)$ and a target sufficiency $\mathcal{R}_t$
  while $RF_t(G) > \mathcal{R}_t$ **do**
    build the dating DAG G_d^t
    compute the dating function
    $active \leftarrow \{\}$
    for all value u in increasing order of definition points **do**
      ExpireOldValues$\big(dp(u)\big)$
      add u to $active$, sorted by increasing killing points
      if $length(active) > \mathcal{R}_t$ **then**
        Spill$\big(dp(u), length(active) - \mathcal{R}_t\big)$
      end if
    end for
  end while

Algorithm 3 Expire Old Values

Require: a definition point i.
  for all value $v \in active$ in increasing order of killing points **do**
    if $kp(v) < i$ **then**
      remove v from $active$
    else
      return
    end if
  end for

Algorithm 4 Spill

Require: a definition point i and the number m of values to be spilled.
  for $l = 1$ to m **do** {spill the m last killed values}
    $s =$ last value in $active$
    if $kp(s) > i$ **then**
      remove s from active
      insert $store(s)$ in G
      insert a flow $e = (s, store)$ with $\delta(e) = lat(s)$
      for all $v \in Cons(s^t)$ **do** {insert a load for each read}
        remove the flow $(u, v) \in E_{R,t}$
        insert $load(s)$ in G
        insert a flow $e = (load, v)$ with $\delta(e) = lat(load)$
        insert an arc $(store, load)$ with $\delta(e) = lat(store)$
      end for
    end if
  end for

Chapter 5

Schedule Independent Register Allocation

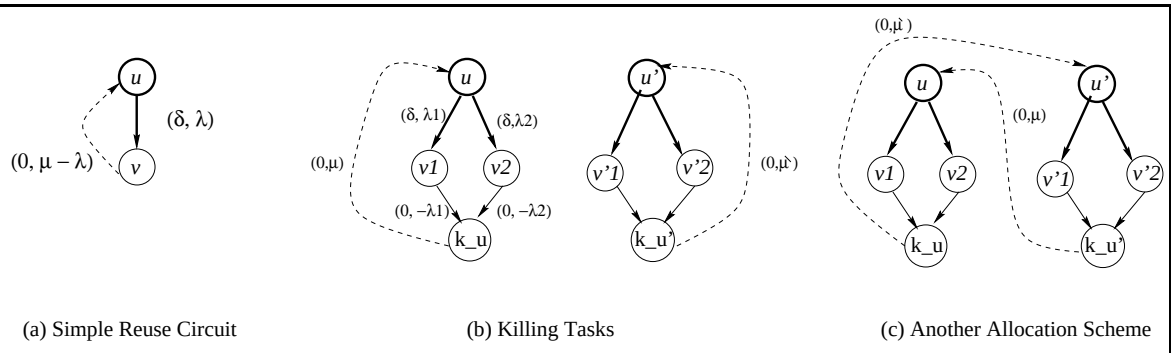
Register allocation in loops is generally performed after or during SWP process. This is because when doing a conventional register allocation in the first step, there is no information of interferences between values live ranges. Consequently, the register allocator introduces an excessive amount of false dependences which dramatically reduces the intrinsic ILP. In this chapter, we present our work [TE02, TE01] that gives a new formulation to perform cyclic register allocation before the scheduling process, directly on the data dependence graph by inserting anti-dependences(*reuse* arcs). This graph extension is first constrained by minimizing the critical circuit and hence minimizing the ILP loss due to register pressure. The second constraint is to ensure that there is always a cyclic register allocation with the set of available registers, and this for any software pipelining of the new graph. We give an exact formulation of this problem with integer linear programming. We also show how our method can be applied when a rotating register file is present. We prove that, in some cases, optimal cyclic register allocation becomes a polynomial problem. Experimental results show that our methods are efficient.

This chapter is organized as follows. We start by a motivating example in Section 5.1 to introduce our ideas for minimal register allocation sensitive to ILP. Then, we formalize the problem by reuse graphs in Section 5.2. We show the tradeoff between register requirement, parallelism and loop unrolling. We provide an exact method by integer programming in Section 5.4. The DDG that we generate has the property that its cyclic register saturation is equal to its cyclic register sufficiency. In the presence of a rotating register file, loop unrolling is not necessary to perform a cyclic register allocation. We extend our formulation in order to take into account this hardware feature in Section 5.5. While the general problem of optimal register allocation under a fixed critical circuit is NP-complete, Section 5.6 presents the cases where this problem becomes polynomial. Before concluding, Section 5.7 details our experiments.

5.1 Motivating Example

The starting point is based on the following idea. Let us consider a flow dependence between u and v of distance λ. This means that the operation v reads the value produced by u λ iterations earlier. Hence, if we use μ different registers cyclically for carrying this value, $u(1)$ and $u(\mu+1)$ store their results in the same register R_1 that will be read subsequently by respectively $v(\lambda+1)$ and $v(\lambda+\mu+1)$. This means that u reuses the same register used by itself μ iterations earlier, and hence creates an anti-dependence between $v(\lambda+1)$ and $u(\mu+1)$ with a distance $\mu-\lambda$. Figure 5.1.(a) is an illustration in which values are shown with bold circles and flow arcs with

bold lines. Dashed ones represent anti-dependences. Since u has a delay to write into the register, the latency of this anti-dependence is set to $-\delta_{w,t}(u)$. This anti-dependence must in turn be counted when computing the new minimum initiation interval $MII \geq \left\lceil \frac{\delta - \delta_{w,t}(u)}{\mu} \right\rceil$.



(a) Simple Reuse Circuit (b) Killing Tasks (c) Another Allocation Scheme

Figure 5.1: Examples of Register Reuse Schemes

More generally, if an operation v kills some register R that is subsequently reused by u μ iterations later (and no other operation uses this register in between), then there is an anti-dependence created between v and u of distance μ.

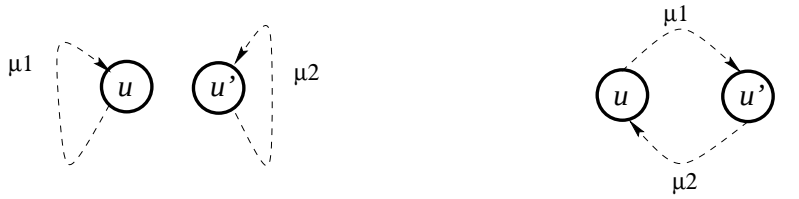
When an operation creates a value that is read by more than one operation, we cannot know in advance which of these consumers would actually kill the value (which one would be scheduled to be the last reader), and hence we cannot know in advance when a register is freed. We propose a trick which defines for each value u^t of type t a virtual killing task k_{u^t}. We insert an arc from each consumer $v \in Cons(u^t)$ to k_{u^t} to reflect the fact that this killing task is scheduled after every (the last) scheduled consumer, see Figure 5.1.(b). The latency of this serial arc is $\delta_{r,t}(v)$, and we set its distance to $-\lambda$ where λ is the distance of the flow dependence between u and its consumer v. We choose this negative or null distance to reflect the fact that the operation $k_{u^t}(i + \lambda - \lambda)$, i.e. $k_{u^t}(i)$, is the virtual killer of $u^t(i)$. Since k_{u^t} is a fictitious task, we could have alternatively considered the positive distance $\max_{e \in E} \lambda(e) - \lambda(u, v)$, which is only a retimed distance.

Now, a register allocation scheme consists in defining the arcs of reuse as defined just above. This amounts to define for each u the task v that reuses the same register. We add then an arc from k_{u^t} to v (representing an anti-dependence from the killer of u to v) with a latency $-\delta_{w,t}(v)$ and a distance $\mu_{u,v}$ to be defined. Note that the dummy node $k_{u^t}(i)$ needs not be inserted if u^t has only one consumer (the killer is necessarily this single consumer).

There are three main constraints that the resulting dependence graph must meet. First, the sum of distances along each circuit must be positive, else the scheduling problem could have no solution. Second, the number of registers used by an allocation scheme (decision) is $\sum \mu$ (we prove this assertion in next section) and must be less than or equal to the number of available registers. Lastly, a register released by an operation can be reused by only one operation, and each operation reuses only one register. This means that the added reuse arcs between the killing nodes and the values must be a bijection. Note that we may have more than one choice

for an allocation decision. For instance, Figure 5.1.(b) gives a situation in which each value reuses the register released by itself. Figure 5.1.(c) is another allocation decision where each value reuses the register released by the other value. This third hypothesis is not mandatory since we can consider n-periodic register allocation. That is, we can first unroll the loop and then we apply a cyclic register allocation. However, we assume a reuse bijection because it is, first, the one used in practice. Second, it gives simple and elegant results.

The reuse relation between values are described by defining a new graph called *a reuse graph*. Figure 5.2.(a) shows the first reuse decision where for instance u (v respectively) reuses the register used by itself μ_1 (μ_2 respectively) iterations earlier. Figure 5.2.(b) is the second reuse choice in which u (v respectively) reuses the register used by v (u respectively) μ_1 (μ_2 respectively) iterations earlier. The resulted data dependence graph after adding killing tasks and anti-dependences (Figure 5.1) to apply register reuse decisions is called the *the DDG associated with a reuse graph*: Figure 5.1.(b) is the DDG associated with Figure 5.2.(a), and Figure 5.1.(c) is the one associated with Figure 5.2.(b). In the next section, we give a formal definition and modeling to the register allocation problem based on reuse graphs.



(a) First Reuse Graph

(b) Second Reuse Graph

Figure 5.2: Reuse Graphs

5.2 Reuse Graphs for Register Allocation

A register allocation consists in choosing which operation reuses which released register. We define:

Definition 5.1 (Reuse Relation) *Let $G = (V, E, \delta, \lambda)$ be a DDG. A reuse relation for a register type $t \in \mathcal{T}$ is a bijection between $V_{R,t}$ and $V_{R,t}$ such that $reuse_t(u) = v$ iff the statement v reuses the register of type t released by the statement u. We note also $reuse_t^{-1}(v) = u$. We associate with this relation a reuse distance $\mu_{u,v}^t$ such that the operation $v(i + \mu_{u,v}^t)$ reuses the register of type t released by the operation $u(i)$*

We represent the reuse relation by a graph (see Figure 5.2):

Definition 5.2 (Reuse Graph) *Let $G = (V, E, \delta, \lambda)$ be a DDG and $reuse_t$ a reuse relation of a register type $t \in \mathcal{T}$. The reuse graph $G^r = (V_{R,t}, E_r, \mu)$ is defined by:*

$$E_r = \{e = (u^t, v^t) \,/\, reuse_t(u) = v \ \wedge \ \mu_t(e) = \mu_{u,v}^t\}$$

We call each arc in a reuse graph G^r a *reuse arc* , and each path in G^r a *reuse path.* Note that we have some similarities between reuse graphs and meeting graphs: each circuit decomposition of a meeting graph corresponds to a reuse graph. However, meeting graphs consider already scheduled loops. A statement u reuses the register freed by a statement v in a MG decomposition iff their circular lifetime intervals meet at a certain clock cycle. We do not have this restriction is reuse graphs, since a reuse arc from u to v only means that the lifetime interval of $u^t(i)$ is before the lifetime interval of $v^t(i+\mu^t_{u,v})$. The further scheduler is let free to schedule $u^t(i)$ and $v^t(i+\mu^t_{u,v})$ so that they do not meet.

Lemma 5.1 *Each reuse path P constructed by inserting all the successive nodes u_i, u_{i+1} with the property that:*

$$reuse_t(u_i) = u_{i+1} \Longrightarrow u_{i+1} \in P$$

is an elementary circuit which we call a reuse circuit. Also, all the reuse circuits of G^r are disjoined:

$$\forall C \neq C' \text{ two reuse circuits} \qquad C \cap C' = \phi$$

Proof:

Since the reuse relation is a bijection, it assigns to each node u_i a unique $reuse_t(u_i) = u_{i+1}$. It exists then a unique arc leaving u_i and entering u_{i+1}.
Let us construct a maximal path $P = (u_0, u_1, \cdots, u_n)$ such that $reuse_t(u_i) = u_{i+1}$ and any other node u in a reuse relation with a node in P necessarily belongs to P:

$$\forall u^t \in V_{R,t} \qquad \exists u_i \in P \,/ reuse_t(u_i) = u \Longrightarrow u \in P$$

The path P is necessarily an elementary circuit because:

1. if $reuse_t(u_n) = u_0$, then there is an arc from u_n to u_0, see Figure 5.3.(a);
2. if $reuse_t(u_n) = u_k \neq u_0$, then u_k has two predecessors in the circuit ($reuse_t^{-1}(u)$ has two values), which is impossible because the reuse relation is a bijection, see Figure 5.3.(b).

Any two reuse circuit C and C' cannot share the same node u. Otherwise u has two successors i.e. $reuse_t(u)$ has two values which is impossible because the reuse relation is a bijection, see Figure 5.3.(c).

⌟

We note $\mathcal{C}$ the set of all the reuse circuits of G^r.

Lemma 5.2 *Let $G^r = (V_{R,t}, E_r, \mu)$ be a reuse graph according to a reuse relation $reuse_t$. Then, any value $u^t \in V_{R,t}$ of a register type $t \in \mathcal{T}$ belongs to a unique reuse circuit C in G^r.*

Proof :

It is a direct consequence of Lemma 5.1. Since reuse circuits are elementary, a value u^t cannot belong to more than one reuse circuit. Furthermore, each value belongs to at least one reuse circuit because the reuse relation is a bijection.

⌟

Let $\mu_t(G^r)$ be the sum of all reuse distances between values of type t :

$$\mu_t(G^r) = \sum_{e=(u,v)\in E_r} \mu^t_{u,v}$$

and we note also $\mu_t(C)$ the sum of all the reuse distances between values of type t which belong to the reuse circuit C :

$$\forall C \text{ a reuse circuit in } G^r : \qquad \mu_t(C) = \sum_{e=(u,v)\in C} \mu^t_{u,v}$$

To report register reuse decisions in the DDG, we have to ensure that if $reuse_t(u) = v$ with a distance $\mu^t_{u,v}$ then $u^t(i)$ must be killed before the definition of $v^t(i + \mu^t_{u,v})$. For this purpose, we define for each value u^t of type t a virtual killing task k_{u^t} which corresponds to its killing date. We insert an anti-dependence arc between k_{u^t} and v iff $reuse_t(u) = v$. The distance of this anti-dependence is set to $\mu^t_{u,v}$.

Definition 5.3 (Killing Node) *Let $G = (V, E, \delta, \lambda)$ be a DDG and $\mathcal{T}$ a set of registers types. A killing node k_{u^t} of a value $u^t \in V_{R,t}$ of type t is a virtual statement that corresponds to the killer of u^t. It is defined by inserting in the DDG G the node k_{u^t} for all $u^t \in V_{R,t}$ as follows :*

- *we add a serial arc $e = (v, k_{u^t})$ from each consumer $v \in Cons(u^t)$ to k_{u^t} ;*
- *for each inserted arc $e = (v, k_{u^t})$, we set its latency to $\delta(e) = \delta_{r,t}(v)$, and its distance to $\lambda(e) = -d$ such that d is the distance of the flow dependence from u to v through a register of type t : $d = \lambda(e')$ with $e' = (u, v) \in E_{R,t}$.*

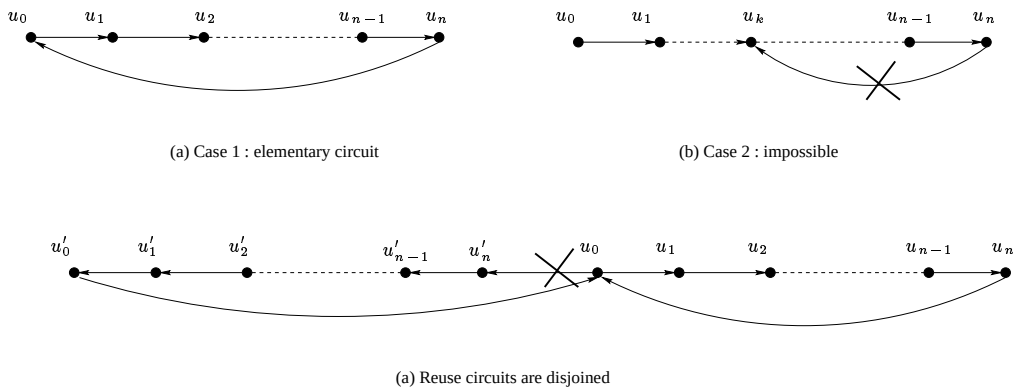


Figure 5.3: Elementary and Disjoined Reuse Circuits

The negative distance inserted from each consumer to the killing task virtually model the fact that u^t and k_{u^t} belong to the same itération i.

Note that the distance in terms of iterations of the path between each value and its killer is null. The set of all killing nodes of type t is noted K_t :

$$K_t = \{k_{u^t} \ / \ u^t \in V_{R,t}\}$$

The resulted data dependence graph after adding the killing tasks and the anti-dependences arcs is called the *DDG associated with the reuse relation.*

Definition 5.4 (DDG associated with a Reuse Relation) *Let $G = (V, E, \delta, \lambda)$ be a DDG with its inserted killing nodes K_t. The DDG associated with a reuse relation $reuse_t$ of a register type $t \in \mathcal{T}$ is an extended DDG of G such that we add an arc $e = (k_{u^t}, v)$ iff $reuse_t(u) = v$. We set its latency to $\delta(e) = -\delta_{w,t}(v)$, and its distance to $\lambda(e) = \mu^t_{u,v}$ (to be defined).*

Parts (b) and (c) of Figure 5.1 are two examples of the DDGs associated with the reuse relation defined in parts (a) and (b) of Figure 5.2 respectively. We note the DDG associated with the reuse relation as $G_{\rightarrow r}$. One can remark that a reuse arc (u, v) is the counterpart of a path (u, v) in the meeting graph of any software pipelined schedule of $G_{\rightarrow r}$. Any arc (k_{u^t}, v) in $G_{\rightarrow r}$ according to a reuse relation ensures that lifetime interval of the value $u^t(i)$ ends before the definition of the value $v^t(i + \mu^t_{u,v})$:

$$\forall \sigma \in \Sigma(G_{\rightarrow r}) \ : \qquad reuse_t(u) = v \Longrightarrow LT_\sigma(u^t(i)) \prec LT_\sigma(v^t(i + \mu^t_{u,v}))$$

where $\Sigma(G_{\rightarrow r})$ is the set of all valid SWP schedules of the DDG $G_{\rightarrow r}$.

Each reuse circuit has counterparts in $G_{\rightarrow r}$. Each counterpart is called an *image* of the reuse circuit :

$$C = (u_0, \cdots, u_n, u_0) \text{ a reuse circuit} \Longleftrightarrow C = (u_0, u'_0, k^t_{u_0}, \cdots, u_n, u'_n, k^t_{u_n}, u_0) \text{ a circuit in } G_{\rightarrow r}$$

in which u'_i is a consumer of u_i. For instance, the reuse circuit (u, v, u) in Figure 5.2.(b) has an image $(u, v_1, k_u, u', v'_1, k_{u'}, u)$ in Figure 5.1.(c). Note that a reuse circuit may have more than one image in $G_{\rightarrow r}$ because a value may have more than one consumer : for instance, a second image for (u, v, u) in Figure 5.2.(b) is $(u, v_2, k_u, u', v'_2, k_{u'}, u)$ in Figure 5.1.(c).

There are some constraints that a reuse relation must meet in order to be valid : the existence of at least a software pipelined schedule for $G_{\rightarrow r}$ (i.e. any introduced circuit must have a positive distance) defines the validity condition of the reuse relation.

Definition 5.5 (Valid Reuse Relation) *Let $G_{\rightarrow r}$ be a DDG associated with a reuse relation $reuse_t$. We say that $reuse_t$ is valid iff $G_{\rightarrow r}$ is schedulable, i.e. there exists a distance $\lambda(e) = \mu^t_{u,v}$ for each arc $e = (k_{u^t}, v)$ with the property that :*

$$\Sigma_L(G_{\rightarrow r}) \neq \phi \Longleftrightarrow \forall \text{ circuit } C \text{ in } G_{\rightarrow r} \quad \lambda(C) > 0 \vee (\lambda(C) = 0 \wedge \delta(C) \leq 0)$$

Figure 5.4 shows two examples of DDGs associated with valid reuse relations. Note that the case of an image of a reuse circuit with a null distance and a negative latency ($\lambda(C) = 0 \ \wedge \ \delta(C) \leq 0$) cannot exist because the anti-dependences can never create a reuse circuit with

a null distance, otherwise it means that an operation (statement instance) reuses the register used by itself, which is absurd. However, a valid reuse relation may introduce a null circuit with a negative latency among other arcs of the graph. We shall see later (Section 5.4.1) how we handle this case.

If a reuse relation is valid, we can build a cyclic register allocation for its DDG associated DDG, as explained in the following theorem.

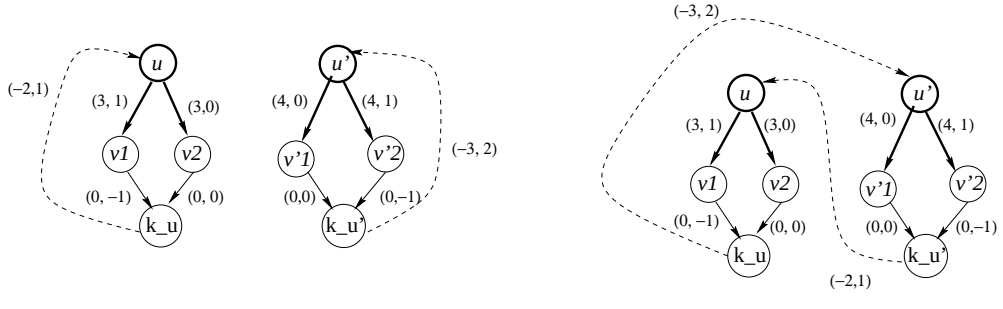
Theorem 5.1 *Let $G_{\rightarrow r}$ be a reuse DDG associated with a valid reuse relation $reuse_t$ such that there is only one reuse circuit in the reuse graph G^r. Then the unique reuse circuit C defines a cyclic register allocation for $G_{\rightarrow r}$ with exactly $\mu_t(C)$ registers if we unroll the loop $\rho = \mu_t(C)$ times.*

Proof :

Let us unroll $G_{\rightarrow r}$ $\mu_t(C)$ times : each statement $u \in V$ has now $\rho = \mu_t(C)$ copies in the unrolled loop. We note u_i the i^{th} copy of the statement $u \in V_{R,t}$. To prove this theorem, we explicitly express the cyclic register allocation, directly on $G_{\rightarrow r}$ after loop unrolling, i.e. we assign registers to the statements of the new loop body (after unrolling). We consider two cases :

Case 1 : all the μ distances are positive For the clarity of this proof, we illustrate it by the example of Figure 5.5 which builds a cyclic register allocation with 3 registers for Figure 5.4.(b) : we have unrolled this loop 3 times. We allocate $\mu_t(C) = 3$ registers in the unrolled loop as explained in Algorithm 5.

1. We choose an arbitrary value u^t in $V_{R,t}$. It has ρ distinct copies in the unrolled loop. So, we allocate ρ distinct registers to these copies. We are sure that such values exist in the unrolled loop body because $\rho > 0$.
2. Since the reuse relation is valid, we are sure that for each reuse arc (u, v), the killing date of an operation $u^t(i)$ is scheduled before the definition date of $v^t(i + \mu^t_{u,v})$. So, we allocate, in the unrolled loop body, the same register of



(a) First Valid Reuse Scheme (b) Second Valid Reuse Scheme

Figure 5.4: Valid Reuse Relations

type t to $v_{\left((i+\mu^t_{u,v}) \bmod \rho\right)}$ as the one allocated to u_i. For instance in Figure 5.5, we allocate the same register R_1 to u_1 and $u'_{((1+2) \bmod 3)} = u'_0$. We are sure that $v_{\left((i+\mu^t_{u,v}) \bmod \rho\right)}$ exists in the unrolled loop body because $\mu^t_{u,v} \geq 0$.

3. We follow the other reuse arcs to allocate the same register to the two values v_i and $v'_{\left((i+\mu^t_{u,v}) \bmod \rho\right)}$ iff $reuse(v) = v'$. We continue in the reuse circuit image until all values in the loop body are allocated.

Since the original reuse circuit image in duplicated ρ times in the unrolled loop, and since each reuse circuit image in the unrolled loop consumes one register, we use in total $\rho = \mu_t(C)$ registers. Dashed lines in Figure 5.5 represent anti-dependences with their corresponding distances after the unrolling.

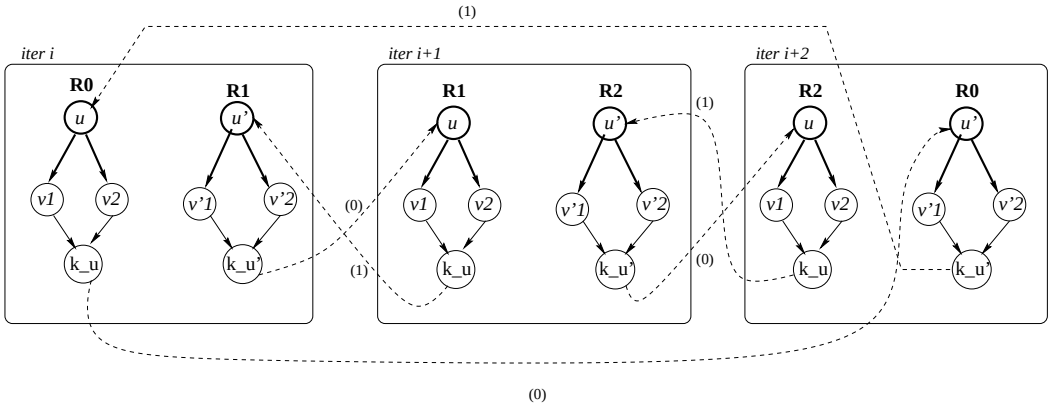


Figure 5.5: Cyclic Register Allocation with One Reuse Circuit

Case 2 : there exists a negative μ distance Here, we cannot express the cyclic allocation directly in the DDG as in the previous case. This is because we cannot consider backward computing (all the operations must be in a lexicographic positive order). However, this does not prevent us for building a register allocation at all. For this purpose, we change the distances of the anti-dependences by using the retiming technique.

A valid retiming, as explained in Chapter 2, makes positive all the distances of the transformed graph, while preserving the same scheduling problem. So, we aim to build a transformed graph from $G_{\rightarrow r}$ which contains positive distances in order to come back to the first case.

Building a valid retimed graph from $G_{\rightarrow r}$ is obvious. Since the reuse relation is valid, then we can build a periodic schedule $\sigma([rn], [cn], h)$ for $G_{\rightarrow r}$. We simply take the retiming function $r(u) = cn(u)$ as explained in [DH00]. The distances become:

$$\forall e = (u, v) \in E \qquad \lambda_r(e) = cn(v) - cn(u) + \lambda(e)$$

Algorithm 5 Cyclic Register Allocation

Require: a DDG $G_{\rightarrow r}$ associated to a valid reuse relation $reuse_t$
  unroll it $\rho = \mu_t(C)$ times {this create ρ copies for each statement}
  for all $u \in V_{R,t}$ **do**
    for all u_i in the unrolled DDG **do** {each copy of u}
      alloc(u_i)$\leftarrow \perp$ {initialization}
    end for
  end for
  choose $u \in V_{R,t}$
  for all u_i in the unrolled DDG **do** {each copy of u}
    alloc(u_i) $\leftarrow$ ListOfAvailableRegisters.pop()
    $n \leftarrow u_i$
    $n' \leftarrow v_{(i+\mu^t_{u,v})mod\ \rho}$ {where $reuse(u) = v$}
    while alloc(n')$=\perp$ **do**
      alloc(n')$\leftarrow$alloc(n)
      $n \leftarrow n'$
      $n' \leftarrow n''$ {where $(k_{n'}, n'')$ is an anti-dependence in the unrolled loop}
    end while
  end for

The dependence constraints are still satisfied:

$$\begin{array}{ll} \forall e = (u,v) & \sigma(v, i+\lambda) - \sigma(u) \geq \delta(e) \\ \forall e = (u,v) & rn(v) - rn(u) + h(cn(v) - cn(u) + \lambda(e)) \geq \delta(e) \\ \forall e = (u,v) & h(cn(v) - cn(u) + \lambda(e)) \geq \delta(e) - rn(v) + rn(u) > -r(v) > -h \end{array}$$

which implies $\lambda_r(e) = cn(v) - cn(u) + \lambda(e) \geq 0$ and $rn(v) \geq \delta(e) + rn(u)$ if $\lambda_r(e) = 0$. I.e. the inter-motif dependences are satisfied while the intra-motif ones become loop carried (satisfied by the successive execution of the iterations).

Finally, since all the distances of the retimed graph are now positive, we refer to the first case to build a cyclic register allocation.

⌟

Note that we can also build a cyclic register allocation with exactly $\mu_t(C)$ registers if we unroll the loop $k \times \rho$ times, in which $\rho = \mu_t(C)$ and $k \in \mathbb{N}^+$, as follows:

1. unroll the loop ρ times and build a cyclic register allocation with $\mu_t(C)$ registers as explained in Theorem 5.1;
2. unroll the allocated loop k times.

If more than one reuse circuit exist, we state in the following theorem that the set of all reuse circuits defines a cyclic register allocation with $\mu_t(G^r)$ registers.

Theorem 5.2 *Let $G_{\rightarrow r}$ be a reuse DDG according to a valid reuse relation $reuse_t$ of a register type $t \in \mathcal{T}$. Then the reuse graph G^r defines a cyclic register allocation for $G_{\rightarrow r}$ with exactly $\mu_t(G^r)$ registers of type t if we unroll the loop α times where :*

$$\alpha = lcm(\mu_t(C_1), \cdots, \mu_t(C_n))$$

in which $\mathcal{C} = \{C_1, \cdots, C_n\}$ is the set of all reuse circuits.

Proof :

It is a direct consequence of Theorem 5.1. The cyclic register allocation is built as follows :

1. unroll the loop α times ; each reuse circuit C has $\frac{\alpha}{\mu_t(C)}$ images in the unrolled loop ;
2. build a cyclic register allocation for each reuse circuit image as explained in Theorem 5.1.

Figure 5.6 is an example of a cyclic register allocation of Figure 5.4.(a) which contains two reuse circuits ; (u, u) with a distance 1, and (u', u') with a distance 2. The unrolling degree is hence $lcm(1, 2) = 2$. The dashed lines represent the anti-dependences after unrolling the loop.

⌟

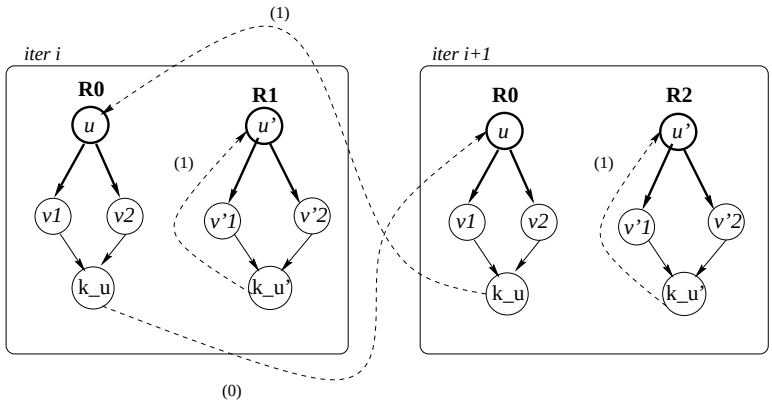


Figure 5.6: Cyclic Register Allocation

Corollary 5.1 *Let $G_{\rightarrow r}$ be a reuse DDG according to a valid reuse relation $reuse_t$ of a register type $t \in \mathcal{T}$. Then, there exists a software pipelined schedule for $G_{\rightarrow r}$ that needs less or equal registers than the number of allocated ones :*

$$\exists \sigma \in \Sigma(G_{\rightarrow r}) : \qquad CRN_t^{\sigma}(G_{\rightarrow r}) \leq \mu_t(G^r)$$

Proof :

According to Theorem 5.2, we can build a valid cyclic register allocation with $\mu_t(G^r)$ available registers. Then, there exists a software pipelined schedule that does not require more than $\mu_t(G^r)$ registers.

⌟

Corollary 5.2 *Let $G = (V, E, \delta, \lambda)$ be a loop with a set of register types $\mathcal{T}$. To each type $t \in \mathcal{T}$ is associated a valid reuse relation $reuse_t$ with its reuse graph. The loop can be allocated with $\mu_t(G^r)$ registers for each type t if we unroll it α times, where*

$$\alpha = lcm(\alpha_{t_1}, \cdots, \alpha_{t_n})$$

in which α_{t_i} is the unrolling degree of the reuse graph for the register type t_i.

Proof:

Direct consequence of Theorem 5.2. The cyclic register allocation is built as follows:

1. unroll the loop α times; each reuse circuit image C_t of register type t in the original loop is duplicated $\alpha_t \times \frac{\alpha}{\mu_t(C)}$ times in the unrolled loop;
2. build a cyclic register allocation for each reuse circuit image of each register type t as explained in Theorem 5.2.

⌟

The next section presents an exact formulation of SIRA by integer programming.

5.3 SIRA Problem Formulation

From the previous section, we deduce that doing a cyclic register allocation of a DDG is equivalent to finding a valid reuse relation. The formal definition of Schedule Independent Register Allocation (SIRA) is:

Problem 5.1 (SIRA) *Let $G = (V, E, \delta, \lambda)$ be a loop and $\mathcal{R}_t$ the number of available registers of type t. Find a valid reuse relation $reuse_t$ such that the corresponding reuse graph $G^r = (V_{R,t}, E_r, \mu)$ has*

$$\mu_t(G^r) \leq \mathcal{R}_t$$

in which the critical circuit in $G_{\rightarrow r}$ is minimized.

Theorem 5.3 *SIRA is NP-complete.*

Proof:

The SIRA decision problem can be formulated as:

Problem 5.2 (dec(SIRA)) *Let $G = (V, E, \delta, \lambda)$ be a loop DDG, $\mathcal{R}_t$ the number of available registers of type t, and k a positive integer. Does there exist a valid reuse relation $reuse_t$ such that*

$$\mu_t(G^r) \leq \mathcal{R}_t$$

and $MII \leq k$.

First, dec(SIRA) belongs to NP :

- to check if a reuse relation is valid, we check if the sum of distances of the reuse circuits are all strictly positive. The set of all the reuse circuits is simply done by looking for strongly connected components of the reuse graph because the reuse circuits are elementary and disjoined ;
- the tests
$$\mu_t(G^r) \leq \mathcal{R}_t$$
and $MII \leq k$ are linear.

Second, dec(SIRA) does not belong to P since it can be easily reduced to the problem of register allocation with a minimum number of registers under a fixed critical path, proven NP-complete in [EGS95].

⌟

5.4 Exact SIRA Modeling

In this section, we give an intLP model for solving SIRA. It is built for a fixed execution rate h. We write linear constraints that define a reuse relation for each register type.

Basic Variables

- a schedule variable σ_u for each operation $u \in V$ including one for each killing node k_{u^t} ;
- a binary variable $\theta^t_{u,v}$ for each $(u,v) \in V^2_{R,t}$ and for each register type $t \in \mathcal{T}$ which is set to 1 iff $reuse_t(u) = v$;
- $\mu^t_{u,v}$ for reuse distance for all $(u,v) \in V^2_{R,t}$.

Linear Constraints

Cyclic Scheduling Constraints

- We bound the scheduling variables (we assume a worst schedule time of one iteration)
$$\forall u \in V \ : \qquad \underline{\sigma_u} \leq \sigma_u \leq \overline{\sigma_u}$$
- data dependences
$$\forall e = (u,v) \in E \ : \qquad \sigma_u + \delta(e) \leq \sigma_v + h \times \lambda(e)$$
- schedule killing nodes for consumed values : $\forall u^t \in V_{R,t}$,
$$\forall v \in Cons(u^t) \,/\, e = (u,v) \in E_{R,t} \ : \qquad \sigma_{k_{u^t}} \geq \sigma_v + \delta_{r,t}(v) + \lambda(e) \times h$$

- if $reuse_t(u) = v$ then there is an anti-dependence between u^t's killer and v. We add an arc from k_{u^t} to v : $\forall t \in \mathcal{T},\ \forall (u,v) \in V^2_{R,t}$:

$$\theta^t_{u,v} = 1 \Longrightarrow \sigma_{k_{u^t}} - \delta_{w,t}(v) \leq \sigma_v + h \times \mu^t_{u,v}$$

  Since $\theta^t_{u,v}$ is binary, we write in the model the following linear constraints :

$$\forall t \in \mathcal{T},\ \forall (u,v) \in V^2_{R,t} : \qquad \theta_{u,v} \geq 1 \Longrightarrow \sigma_{k_{u^t}} - \delta_{w,t}(v) \leq \sigma_v + h \times \mu^t_{u,v}$$

  We use the linear expression of implication defined in [Tou01d, Tou01c, Tou01a].

- If there is no register reuse between two values ($reuse_t(u) \neq v$), then $\theta^t_{u,v} = 0$. The anti-dependence distance $\mu^t_{u,v}$ must be set to 0 in order to not be cumulated in the objective function. $\forall t \in \mathcal{T},\ \forall (u,v) \in V^2_{R,t}$:

$$\theta^t_{u,v} = 0 \Longrightarrow \mu^t_{u,v} = 0$$

Reuse Relation Constraints The reuse relation must be a bijection :

- a register can be reused by only one operation :

$$\forall t \in \mathcal{T},\ \forall u \in V_{R,t} : \qquad \sum_{v \in V_{R,t}} \theta^t_{u,v} = 1$$

- one value can reuse only one released register :

$$\forall t \in \mathcal{T},\ \forall u \in V_{R,t} : \qquad \sum_{v \in V_{R,t}} \theta^t_{v,u} = 1$$

Objective Function We want to minimize the number of registers required for register allocation. So, we choose an arbitrary register type t that we use as an objective function :

$$\text{Minimize} \sum_{(u,v) \in V^2_{R,t}} \mu^t_{u,v}$$

This function is necessarily positive, since all reuse circuit images have $\sum \mu > 0$. Other register types are bounded in the model by their respective number of available registers :

$$\forall t' \in \mathcal{T} - \{t\} : \sum_{(u,v) \in V^2_{R,t'}} \mu^{t'}_{u,v} \leq \mathcal{R}_{t'}$$

Summary The reuse relation produced is necessarily valid since we succeed in constructing a cyclic schedule. The complexity of the model is bounded by $\mathcal{O}(|V|^2)$ variables and by $\mathcal{O}(|E| + |V|^2)$ constraints. To solve SIRA, we proceed as follows.

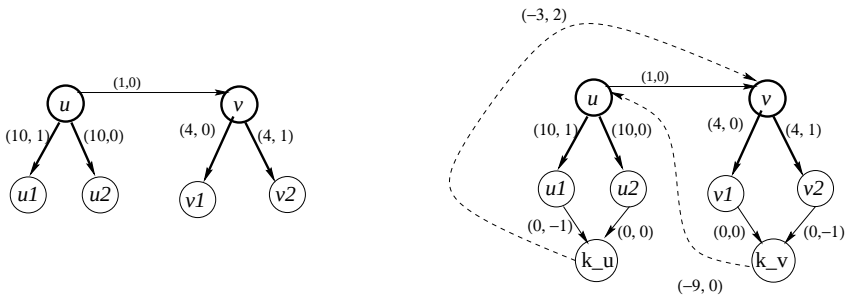
1. We start by solving an intLP with $h = MII$.
2. If the solution is greater than $\mathcal{R}_t$, then we increment h (a dichotomy between h and a maximum $h_{max} = L$).
3. If we reach the maximum h_{max} without finding a solution, then there is not a cyclic register allocation with R_t registers. Therefore, spill code must be introduced (see Section 4.2).

In some cases, an optimal SIRA solution must introduce null circuits to the constructed DDG so as to have a minimal number of allocated registers. The next section discusses this problem.

5.4.1 Problem of Circuits with Null Distances

Our loop model admits explicit writing delays for statements. So, some anti-dependence arcs in $G_{\rightarrow r}$ may have negative latencies. If we do not take care during the computation of an optimal register allocation (minimizing the register requirement under a fixed execution rate), the produced DDGs according to the computed reuse relation may contain a null circuit with a negative latency. Even if such a graph is schedulable, we cannot admit it since we cannot ensure that is would remain schedulable in the presence of resource constraints.

Figure 5.7 is an illustration. In the original loop shown in Part (a), there exists a dependence path from u to v with a null distance (the path is in the loop body). A valid reuse relation as shown in Part (b) may assign the same register to $u(i)$ and $v(i)$ by fixing $reuse_t(v) = u$. This creates an anti-dependence from $v(i)$'s killer to $u(i)$. Since the latency of the reuse arc (k_v, u) is negative (-9) and the latency of the path $u \rightsquigarrow k_v$ is 5, the null circuit (v, k_v, u, v) does not prevent the associated DDG from being modulo scheduled but may be so in the presence of resource constraints. In this section, we show how we include new constraints in the exact SIRA modeling to avoid this disadvantage.



(a) Original Loop

(b) DDG associated to a valid reuse relation

Figure 5.7: Null Circuits produced by SIRA

To eliminate optimal SIRA solutions that require null distance circuits, we can use two solution. A first one is to not introduce anti-dependences with negative or null latencies. This is done by considering sequential semantic for register usage, i.e. by setting in the intLP model $\delta_{r,t} = 0$ and $\delta_{w,t} = -1$. This technique remains optimal in the case of sequential superscalar codes, but may be sub-optimal in static issue (VLIW) codes. An optimal solution is given below.

A second solution is to guarantee the existence of a topological sort for the loop body of the constructed DDG $G_{\rightarrow r}$. However, since the constructed DDG may contain arcs with negative distances, we may not be able to detect some circuits with null distances. We then have to work on a retimed graph so as to have only positive distances. Then, each arc with a null distance in the retimed graph is an arc in the loop body. If we guarantee that there is no null distance circuit in the retimed graph, then the non retimed DDG does not contain a null circuit (and vice versa). Hence, the intLP model constructs a retimed graph $G'_{\rightarrow r}$. All the anti-dependences are positive in the retimed graph $G'_{\rightarrow r}$, but can be negative in the non transformed DDG $G_{\rightarrow r}$.

The intLP system is slightly modified so as to include retiming and topological sort constraints as follows.

- The objective function remains the same, since the number of allocated registers in a reuse circuit is not modified by loop retiming :

$$\text{Minimize} \sum_{(u,v)\in V_{R,t}^2} \mu_{u,v}^t$$

- The integer variables are the following.
  1. For each node $u \in V$, we define an integer retiming coefficient r_u.
  2. We add the variables of a topological sort. For each statement $u \in V$, we define an integer $d_u \leq |V|$.
- The linear constraints are the following.
  1. In the intLP system of SIRA, we replace each distance $\lambda(e) \,/\, e = (u, v)$ by $(\lambda(e) + r_v - r_u)$. We also replace the anti-dependence distances $\mu_{u,v}^t$ by the retimed distance $(\mu_{u,v}^t + r_v - r_{k_{u^t}})$. Since h is constant in the model, the system remains linear.
  2. The retiming must be valid. We add the following constraints.
     - For each original arc $e = (u, v) \in E$, write :
     $$\lambda(e) + r_v - r_u \geq 0$$
     - For each introduced anti-dependence arc, the retimed distance must be positive. So we write :
     $$\forall u, v \in V_{R,t} \ : \ \theta_{u,v}^t = 1 \Longrightarrow \mu_{u,v}^t + r_v - r_{k_{u^t}} \geq 0$$
  3. We add topological sort constraints as follows.
     - For the original arcs, we write :
     $$\forall e = (u, v) \in E \ : \quad \lambda(e) + r_v - r_u = 0 \Longrightarrow d_u < d_v$$
     - For the introduced anti-dependences, we write :
     $$\forall u, v \in V_{R,t} \ : \ \left.\begin{array}{l} \theta_{u,v}^t = 1 \\ \mu_{u,v}^t + r_v - r_{k_{u^t}} = 0 \end{array}\right\} \Longrightarrow d_{k_{u^t}} < d_v$$

We add at most $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|V|^2 + |E|)$ linear constraints to eliminate optimal solution with null circuits.

The unrolling degree is left free and over any control in our SIRA formulation. The theoretical upper-bound of the unrolling degree required for allocating $\mathcal{R}$ registers is $e^{(1+\mathcal{O}(1))\sqrt{\mathcal{R}\ln\mathcal{R}}}$. This is a classical mathematical problem where, as far as we know, no exact upper-bound has been found yet ![1] Minimizing the unrolling degree amounts to minimize $lcm(\mu_i)$ the least common multiple of the anti-dependence distances of reuse circuits. This problem is very difficult since there is no way to linearly express the least common multiple. We can consider two solutions.

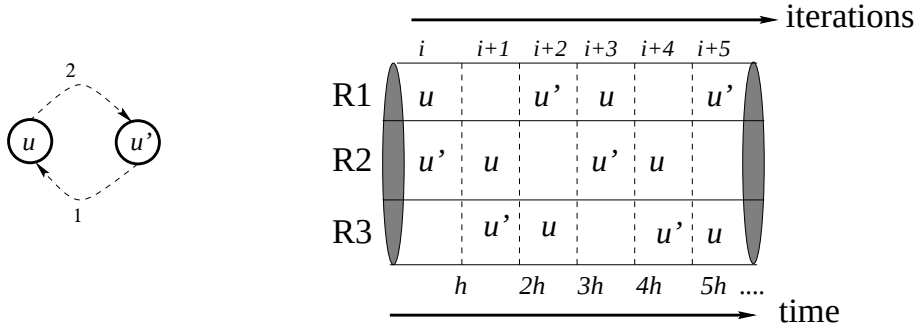
[1] This upper-bound corresponds to the order of the maximal cycle subgroup of permutation group on $\mathcal{R}$ elements [Lan74].

1. We set limits on the reuse distances with strictly positive constants ($\mu_1 \leq c_1, \cdots, \mu_n \leq c_n$). The smaller these constants, the more the unrolling degree is minimized, the more the critical circuit increases while the system becomes more difficult to solve. We think that this solution is non efficient and inaccurate.

2. We look for only one reuse (hamiltonian) circuit : the unrolling degree becomes equal to the number of allocated registers, and hence is minimized by the objective function that minimizes the register requirement. This solution is studied in the next section.

5.5 SIRA with Rotating Register Files

A rotating register file, as explained in Section 2.4.1, is a hardware feature that implicitly moves (shifts) ISA (architectural) registers in a cyclic way. At each new kernel issue (special branch operation), each architectural register specified by a program is mapped by hardware to a new physical register. The mapping function is (R denotes an architectural register and R' a physical register) : $R_i \mapsto R'_{(i+RRB) \bmod s}$ where RRB is a rotating register base and s the total number of physical registers. The index of that physical register is continuously decremented at each new kernel. Consequently, the intrinsic reuse scheme between statements necessarily describes a hamiltonian reuse circuit. The hardware behavior of such register files does not allow other reuse patterns. SIRA in this case must be adapted in order to look for only hamiltonian reuse circuits. Figure 5.8 gives an example to see how a hamiltonian reuse circuit describes a cyclic register allocation on a RRF. Part (b) shows the writing of values in physical registers.



(a) Hamiltonian Reuse Circuit (b) Cyclic Register Allocation on a RRF

Figure 5.8: SIRA with a Rotating Register File

Furthermore, even if no rotating register file exists, looking for a reuse relation with a unique hamiltonian reuse circuit makes the unrolling degree equal to the number of needed registers. The objective function minimizes both of them.

Since a reuse circuit is always elementary (Lemma 5.1), it is sufficient to state that a hamiltonian reuse circuit with $n = |V_{R,t}|$ nodes is a reuse circuit of size n. We proceed by forcing a numbering of the statements from 1 to n according to the reuse relation.

Definition 5.6 (Hamiltonian Ordering) *Let $G = (V, E, \delta, \lambda)$ be a loop and $reuse_t$ a valid reuse relation of a register type $t \in \mathcal{T}$. A hamiltonian ordering ho_t of this loop according to its reuse relation is a function defined by:*

$$\begin{array}{rccl} ho_t & : & V_{R,t} & \rightarrow & \mathbb{N} \\ & & u^t & \mapsto & ho_t(u) \end{array}$$

such that $\forall u, v \in V_{R,t}: \quad reuse_t(u) = v \iff ho_t(v) = \Big(ho_t(u) + 1\Big) \ mod \ |V_{R,t}|$

Figure 5.9 is an example of a hamiltonian ordering of a reuse graph with 5 values.

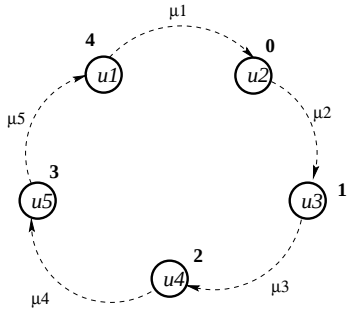


Figure 5.9: Hamiltonian Ordering

Theorem 5.4 *Let $G = (V, E, \delta, \lambda)$ be a loop and $reuse_t$ a valid reuse relation of a register type $t \in \mathcal{T}$. A hamiltonian ordering of the reuse relation makes the reuse graph a hamiltonian graph.*

Proof:

Suppose that the contrary is true, i.e. there exists a hamiltonian ordering such that there is no unique reuse circuit. For simplicity without loss of generality, we suppose that there are two reuse circuits:

$$\exists ho_t \text{ a ham numbering}: \qquad C_1, C_2 \text{ are two reuse circuits in } G^r$$

According to Lemma 5.2 and Lemma 5.1, each value $u^t \in V_{R,t}$ belongs to a unique reuse circuit which is disjoined and elementary. Let $C_1 = (u_1, \cdots, u_m, u_1)$ and $C_2 = (v_1, \cdots, v_k, v_1)$ be two reuse circuits where $m + k = n = |V_{R,t}|$ and $m, k < n$. Since the number of nodes in C_1 is $m < n$, the total number of the hamiltonian orders does not contain n successive integers modulo n:

$$|\{ho_t(u^t) \in \mathbb{N} \,/\, u^t \in C_1\}| < n$$

Let $u_i \in C_1$ be the node with the highest hamiltonian order:

1. if $ho_t(u_i) < n - 1$ then $ho_t(u_{i+1}) = ho_t(u_i) + 1$ (see u_1 in Figure 5.10.a), contradiction (u_i has not the highest hamiltonian order) !

2. $ho_t(u_i) = n-1$ necessarily and $ho_t(u_{i+1}) = 0$. By following the successive nodes in C_1 from u_{i+1} to u_i, there are only $m-1$ nodes and hence $ho_t(u_i) = m-1 < n-1$ necessarily (see u_1 in Figure 5.10.b), contradiction !

Then, a hamiltonian ordering implies necessarily that there exists a unique reuse hamiltonian circuit. One can remark that a hamiltonian ordering is unique.

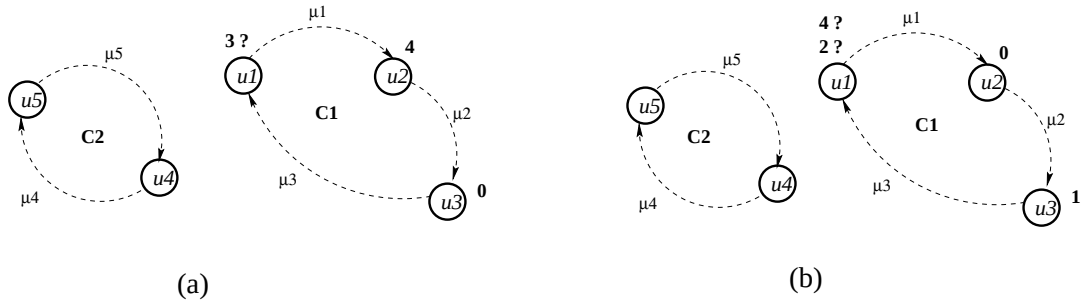


Figure 5.10: Hamiltonian Ordering produces a Hamiltonian Reuse Circuit

⌟

Problem 5.3 (SIRA_HAM) *Let $G = (V, E, \delta, \lambda)$ be a loop and $\mathcal{R}_t$ a positive integer. The SIRA_HAM problem is to find a valid reuse relation $reuse_t$ with a hamiltonian ordering ho_t such that the corresponding reuse graph $G^r = (V_{R,t}, E_r, \mu)$ has*

$$\mu_t(G^r) \leq \mathcal{R}_t$$

and the critical circuit in $G_{\rightarrow r}$ is minimized.

Exact SIRA_HAM Formulation

We add to the intLP model of SIRA (defined in Section 5.4) the variables and linear constraints that define a hamiltonian ordering :

1. for each register type and for each value $u^t \in V_{R,t}$, we define an integer variable ho_{u^t} which corresponds to its hamiltonian ordering ;

2. we include in the model the bounding constraints of the hamiltonian ordering variables :

$$\forall u^t \in V_{R,t} : \qquad ho_{u^t} < |V_{R,t}|$$

3. we define linear constraints of the modulo hamiltonian ordering by including in the model :

$$\forall u, v \in V^2_{R,t} : \qquad \theta^t_{u,v} = 1 \Longleftrightarrow ho_{u^t} + 1 = |V_{R,t}| \times \beta^t_{u,v} + ho_{v^t}$$

where $\beta^t_{u,v}$ is a binary variable that holds to the integer division of $ho_{u^t} + 1$ on $|V_{R,t}|$. We use the linear expression of equivalence previously defined in [Tou01d, Tou01c, Tou01a].

We have expanded the exact SIRA intLP model by at most $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|V|^2)$ linear constraints.

When looking for a hamiltonian reuse circuit, we have some similarities with the problem of finding a hamiltonian circuit in a meeting graph (see Section 2.4 and Theorem 2.5). In the latter case, we may need one extra register to construct such a circuit.

Proposition 5.1 *Hamiltonian SIRA needs at most one more register than SIRA.*

Proof:

It is a consequence of meeting graph results because a MG describes the reuse relation between values. If we choose to decompose a MG into elementary circuits, then an arc (u, v) in the MG corresponds to a reuse arc in the reuse graph. We have two cases.

1. There exists an optimal SWP schedule for G with execution rate h such that its meeting graph has a hamiltonian circuit. Suppose that this schedule needs R registers (optimal MAXLIVE). Then, according to Theorem 2.3, there exists a cyclic register allocation with R registers. If so, SIRA intLP formulation finds $\sum \mu = R$ under a fixed h since it is an exact formulation. Also, according to Theorem 2.5, there exists a hamiltonian circuit in MG, and hence there exists a cyclic register allocation with R registers. Since a hamiltonian reuse circuit exists (image of the MG hamiltonian circuit), hamiltonian SIRA intLP finds $\sum \mu = R$ under a fixed h since it is an exact formulation.
2. There is no optimal SWP schedule for G with execution rate h such that its meeting graph has a hamiltonian circuit. Suppose that this schedule needs R registers (optimal MAXLIVE). Then, according to Theorem 2.3, there exists a cyclic register allocation with R registers. SIRA intLP formulation finds $\sum \mu = R$ under a fixed h since it is an exact formulation. Also, according to Theorem 2.6, there exists a cyclic register allocation with $R+1$ registers with a hamiltonian reuse pattern. Hamiltonian SIRA intLP finds $\sum \mu = R+1$ under a fixed h since it is an exact formulation.

⌟

Both SIRA and hamiltonian SIRA are NP-complete. Fortunately, we have some optimistic results. In next section, we investigate the case in which SIRA can be solved in polynomial time.

5.6 Polynomial Cases for SIRA

In this section, we show that if we fix reuse arcs, i.e. if we fix the register sharing decision among statements, then determining μ distances so as to minimize the register requirement and the critical circuit is solvable in polynomial time. We neglect for the moment the problem

of null circuits (we discuss it later).

Let $G = (V, E, \delta, \lambda)$ be a loop and $G_{\rightarrow r}$ the DDG associated with a reuse relation $reuse_t$ according to a register type t, such that the reuse distances $\mu^t_{u,v}$ are not fixed yet. In the following, we write the integer programming model to solve SIRA. The intLP is considerably simplified and we show that its constraint matrix is totally unimodular.

As we know, the graph $G_{\rightarrow r} = (V_{\rightarrow r}, E_{\rightarrow r}, \delta_{\rightarrow r}, \lambda_{\rightarrow r})$ have the following nodes:

- the set of the nodes V of the original loop G. The set $V_{R,t} \subseteq V_{\rightarrow r}$ is the set of statements writing into the registers of type t;
- the set of killing nodes $k_{u,t}$ for each $u \in V_{R,t}$.

The set of arcs $E_{\rightarrow r}$ contains:

- the set of the arcs E of the original loop G, where $\delta_{\rightarrow r}(e) = \delta(e)$ and $\lambda_{\rightarrow r}(e) = \lambda(e)$ for each $e \in E$. The set $E_{R,t} \subseteq E$ is the set of flow dependences through the values of type t;
- the set of arcs which connect the consumers to the killing nodes $\{e = (v, k_{u^t}) / \ v \in Cons(u^t)\}$, in which
$$\delta_{\rightarrow r}(e) = \delta_{r,t}(v) \quad \text{for} \quad e = (v, k_{u^t})$$
and
$$\lambda_{\rightarrow r}(e) = -\lambda(e) \quad \text{for} \quad e = (u, v) \in E_{R,t}$$
- the set of reuse arcs $e = (k_{u^t}, v)$ for $reuse_t(u) = v$, where $\delta_{\rightarrow r}(e) = -\delta_w(v)$ and the distance $\lambda_{\rightarrow r}(e) = \mu^t_{u,v}$ has to be defined. We note the set of these reuse arcs by $E_r \subseteq E_{\rightarrow r}$. Remember that the reuse relation is a bijection, and hence each value u^t has one and only one reuse arc leaving k_{u^t}, and one and only one reuse arcs entering u. Therefore, $|E_r| = |V_{R,t}|$.

Hence, the intLP system that solve SIRA with fixed reuse arcs is considerably simplified as follows.

$$\begin{array}{ll} \text{Minimize} & \sum_{(k_{u^t},v)\in E_r} \mu^t_{u,v} \\ \\ \text{Subject to:} & \\ h\mu^t_{u,v} + \sigma_v - \sigma_{k_{u^t}} \geq -\delta_w(v) & \forall (k_{u^t}, v) \in E_r \\ \sigma_v - \sigma_u \geq \delta(e) - h\lambda(e) & \forall e = (u, v) \in (E_{\rightarrow r} - E_r) \end{array} \tag{5.1}$$

Since h is a constant, we do the variable substitution $\mu'_u = h \times \mu^t_{u,v}$ and System 5.1 becomes:

$$\begin{array}{ll} \text{Minimize} & \sum_{u\in V_{R,t}} \mu'_u \\ \\ \text{Subject to:} & \\ \mu'_u + \sigma_v - \sigma_{k_{u^t}} \geq -\delta_w(v) & \forall (k_{u^t}, v) \in E_r \\ \sigma_v - \sigma_u \geq \delta(e) - h\lambda(e) & \forall e = (u, v) \in (E_{\rightarrow r} - E_r) \end{array} \tag{5.2}$$

There are $\mathcal{O}(|V_{\rightarrow r}|) = \mathcal{O}(|V|)$ variables and $\mathcal{O}(|E_{\rightarrow r}| = \mathcal{O}(|V| + |E|))$ constraints in this system.

Theorem 5.5 *The constraint matrix of the integer programming model in System 5.2 is totally unimodular, i.e. the determinant of each square sub-matrix is equal to 0 or to $\pm$ 1.*

Proof:

See Appendix A.

⌟

Thanks to Theorem 5.5, we can optimally solve the integer programming model of System 5.2 with a polynomial time method, in which the complexity depends on the size of $V_{\rightarrow r}$ and $E_{\rightarrow r}$.

The case described in this section can be used in practical compilers in different ways. Here are some examples.

1. For each value $u \in V_{R,t}$, we can decide that $reuse_t(u) = u$. This means that each statement reuses the register freed by itself (no sharing of registers between different statements). This is similar to buffer minimization problem as described in [NG93].
2. We can fix reuse arcs according to the anti-dependences present in the original code: if there is an anti-dependence between two statement u and v in the original code, then fix $reuse_t(u') = v$ with the property that u kills u'. This decision is a generalization of the problem of reducing the register requirement as studied in [WKE95]. The authors assumed fixed row numbers and fixed anti-dependencies. Our result shows that the problem is still polynomial for non fixed row numbers.
3. With a rotating register file, we can fix an arbitrary (or with a cleverer method) hamiltonian reuse circuit among statements.

Finally, it remains to eliminate optimal solutions with null circuits.

Null Circuits in Polynomial SIRA As described before, this problem arises for static issue processors (VLIW) with explicit writing offsets. If we add topological sort constraints as described in Section 5.4.1, we lose the property of the constraints matrix of System 5.2 since it becomes not totally unimodular. However, since reuse arcs are fixed statically, we can take some precautions at compile time to avoid null circuits in the optimal solution.

- If there exists a longest path in the loop body $P = u \rightsquigarrow v$ with $\delta(P) \leq \delta_{w,t}(u) - \delta(v \rightsquigarrow k_v)$ and if we have fixed $reuse(v) = u$, we only have to add the bounding information $\mu^t_{v,u} > 0$ which does not alter the total unimodularity of the constraints matrix nor the optimality of the solution. This is a necessary condition to avoid null circuits, but it may not be sufficient, since it eliminates some critical solutions but not all of them. If an optimal solution still generates null circuits, we are in the presence of a more critical problem. We propose to carry on with the following method that is satisfactory for us.
- At this point, the previous constraints do not prevent an optimal solution from adding null circuits in the DDG $G_{\rightarrow r}$. Let C^* be a circuit with null distance in the computed optimal solution. We proceed by iteratively setting lower-bounds on anti-dependence distances so as to avoid null circuits, as follows:
  1. We choose an arbitrary anti-dependence arc (k_u, v) in C^*. Let $\mu^* = \mu^t_{u,v}$ be the computed distance that makes C^* a circuit with null distance.

2. We introduce a new constraint in the intLP system by adding the information :

$$\mu^t_{u,v} > \mu^*$$

Then, we solve the system. This constraint does not alter the total unimodularity of System 5.2, but may produce a sub-optimal solution.

3. We iterate this step until reaching a DDG $G_{\rightarrow r}$ without a null circuit. Since there is only $|V_{R,t}|$ anti-dependence arcs, we iterate at most $|V_{R,t}|$ times before reaching a satisfactory solution.

5.7 Experiments

We have developed a tool to cyclically allocate registers in loops using SIRA. It is based on two underlying softwares.

1. LEDA-4.1 (Library of Efficient Data types and Algorithms [MN99]) from Algorithmic Solutions Software. This library is used for handling the graphs (DDGs, reuse DDGs, etc.) and generating the integer linear programs ;

2. CPLEX-7.0 (see [CPL93]) from Ilog. It is an optimizer for solving linear, mixed-integer and quadratic programming problems.

Our tool uses the two strategies : the classical SIRA in which the reuse circuits are free from any control, and the hamiltonian case where we look for a hamiltonian reuse circuit. We have also developed the polynomial SIRA case as studied in Section 5.6. Two main strategies have been experimented : self reuse arcs where we fix $reuse(u) = u$ for any value, and a fixed hamiltonian reuse circuit. In the latter case, the hamiltonian circuit is arbitrary : we arbitrarily numbered the values from 1 to n and we fixed $reuse(u_i) = reuse(u_{(i+1) \ mod \ n})$.

Our benchmarks are presented in [TT00]. The performance of these loops are bounded by floating point computation. So, we focus on this register type and we assume that we target superscalar codes (null reading and writing delays). This section summarizes our conclusions.

5.7.1 Optimal SIRA

The optimal SIRA solutions are described in Table 5.1. The two main columns correspond to the two SIRA formulations : the first is the "classical SIRA" as explained in Section 5.3, in which the unrolling degree is left free from any constraint, and the second is the hamiltonian SIRA formulation as explained in Section 5.5 intended for both minimizing the unrolling degree (in this case, it is equal to the number of allocated registers) and to the rotating register file. Note that we didn't succeed in finding an optimal solution in three cases because of the computation complexity. We treat the latter cases by using heuristics in a further paragraph..

Table 5.1 shows the minimum number of fp registers required to perform cyclic register allocation if we do not want to increase the critical circuit (no ILP loss) :

1. 64 fp registers are sufficient for all loop ;

2. 32 fp registers are sufficient for 91.66% of loops ;

3. 16 fp registers are sufficient for 91.66% of loops ;

4. 8 fp registers are sufficient for 83.33% of loops ;

5. 4 fp registers are sufficient for 50.00% of loops ;

The difference between the solutions of the two SIRA formulations is shown in Table 5.2. Hamiltonian SIRA needs in the worst case one more register than SIRA (2 cases only). The unrolling degree is kept under control with hamiltonian SIRA since it is equal to the number of registers. However, even if SIRA exhibits better unrolling degrees in most cases, the case of spec-spice-loop7 shows that this factor may grow exponentially if it is left free.

We also have experimented SIRA on these loops with different critical ratios h, starting from MII to L. Figures 5.11 and 5.12 give some of representative results. As expected, the number of registers decreases if we increment the execution rate. The lower the critical circuit is, the higher is the number of registers. In some cases, releasing the critical circuit by only one clock period dramatically decreases the register need : for instance spec-dod-loop7 needs 35 fp registers with a critical circuit $MII = 1$, but needs only 18 fp registers if the critical circuit is $MII = 2$. In other cases, the number of required registers is the same for any critical circuit : for instance spec-dod-loop3 needs 3 fp registers for any execution rate. The optimal solutions for hamiltonian SIRA (not plotted) are in most cases equal to those computed by the "classical" SIRA, except in very few cases where we need one extra register.

Using Heuristics for Solving Optimal SIRA

During our experiments, the solver could not find the optimal solution of some loops because of the problem complexity : the computation space was saturated and CPLEX ran out of memory (remember that the problem is NP-complete). In such cases, we used some heuristics and techniques to get a suitable or approximated solution. Fortunately, CPLEX supports such features.

Table 5.3 describes our SIRA experiments using some of these resolution techniques. The first column gives the results if we stop the optimization process when the number of allocated registers is less than or equal to 16. In the second one, we have set a limit of five minutes to the computation time. In the third, we have limited the work space to 20 mega bytes. Lastly, we have limited the number of integer solutions to 3. As can be seen, we can always use intLP formulation to get an approximated solution.

5.7.2 SIRA with Fixed Reuse Arcs

We have experimented SIRA with fixed reuse arcs (polynomial cases) on all the loops with various initiation intervals. Results are shown in Figure 5.13 to 5.17. Clearly, except in few cases, the self reuse strategy needs the highest number of registers. This is because each value needs at least one register, since we prevent two distinct statements from sharing the same register. So, the minimum number of needed registers with a self reuse strategy is always down-bounded by $|V_R|$ the number of values (statements) in the loop body. This is because each statement needs at least one register (buffer) if no sharing exists. The difference between the registers needed with this strategy and a fixed arbitrary hamiltonian reuse circuit may be deep. An interesting

result is that the number of registers needed when fixing an arbitrary hamiltonian reuse circuit is near to the optimal in many cases. The maximal experimental difference between the register requirement of fixed hamiltonian SIRA with the optimum is 4 registers.

5.7.3 Unrolling Degrees

Figure 5.18 to 5.22 plot the unrolling degrees of all the SIRA strategies : optimal SIRA, optimal hamiltonian SIRA, and the two polynomial cases (self reuse and fixed hamiltonian circuit). While the self reuse strategy needs the highest number of registers, its unrolling degrees exhibit the lowest ones in most cases. This is useful technique if the code size expansion is a critical constraint (as in embedded softwares). In most cases, the unrolling degrees are acceptable (less than the number of allocated registers). Unfortunately, the example of spec-spice-loop7 in Figure 5.20 shows that the unrolling degree may be very high if not kept under control. In this case, using a hamiltonian reuse circuit is better since the objective function minimizes this factor.

5.8 Conclusion

This chapter presents a new approach consisting in building an early cyclic register allocation before code scheduling with multiple register types and delays in reading/writing. Our formulation is based on reuse graphs to model the fact that two statements use the same register as storage location. An intLP model gives optimal solution and enables us to make a tradeoff between ILP loss (increase of MII) and the number of required registers.

Each reuse decision implies loop unrolling with a factor depending on reuse circuits for each register type. Optimizing this factor is a hard problem and no satisfactory solution exists (as far as we know). However, we do not need to unroll in the presence of a rotating register file. We only need to seek a unique hamiltonian reuse circuit. For this purpose, we add new variables and linear constraints to SIRA intLP model that build such circuit by using hamiltonian numbering. The penalty for this hamiltonian circuit constraint is at most one extra register than the optimal for the same MII. Experimental results show that only few cases need this extra register.

While looking for optimal register allocation is NP-complete, fixing reuse arcs and finding the minimal number of required registers can be optimally solved with polynomial algorithms. We can use this result in different ways, as setting self-reuse arcs or fixing an arbitrary (or with a cleverer technique) hamiltonian circuit. Experiments show that self-reuse decision needs the highest number of registers, while fixing an arbitrary hamiltonian reuse circuit needs much less registers. However, unrolling degrees with self-reuse are better.

Our experiments show that performing a minimal register allocation with a self reuse strategy (buffer minimization) isn't a good decision in terms of register requirement. We think that how registers are shared between different statements is one of the most important issues, and preventing this sharing by a self reuse strategy consumes much more registers than needed by other reuse decisions.

		SIRA		Hamiltonian SIRA
Loop	**MII**	**Min R**	**unroll**	**Min R**
Lin-ddot	1	7	2	7
Liv-loop1	4	5	3	5
Liv-loop5	3	3	3	3
Liv-loop23	8	NA	NA	NA
Spec-dod-loop1	13	NA	NA	NA
Spec-dod-loop2	21	3	2	3
Spec-dod-loop3	20	3	2	3
Spec-dod-loop7	1	35	34	35
Spec-fp-loop1	20	2	2	2
Spec-tom-loop1	22	NA	NA	NA
Spec-spice-loop1	1	3	3	3
Spec-spice-loop2	1	15	4	15
Spec-spice-loop3	6	2	2	2
Spec-spice-loop4	10	8	2	9
Spec-spice-loop5	3	1	1	1
Spec-spice-loop6	2	14	11	14
Spec-spice-loop7	1	40	180	40
Spec-spice-loop8	1	7	4	7
Spec-spice-loop9	3	7	4	7
Spec-spice-loop10	3	2	1	2
Whet-cycle4_1	4	1	1	1
Whet-cycle4_2	2	2	1	2
Whet-cycle4_4	1	4	2	4
Whet-cycle4_8	1	8	10	8
Whet-loop1	17	5	2	5
Whet-loop2	6	5	4	6
Whet-loop3	5	4	2	4

Table 5.1: SIRA Solutions

Loop	**Min R**	**unroll**
Lin-ddot	0	5
Liv-loop1	0	2
Liv-loop5	0	0
Spec-dod-loop2	0	1
Spec-dod-loop3	0	1
Spec-dod-loop7	0	1
Spec-fp-loop1	0	0
Spec-spice-loop1	0	0
Spec-spice-loop2	0	11
Spec-spice-loop3	0	0
Spec-spice-loop4	1	7
Spec-spice-loop5	0	0
Spec-spice-loop6	0	3
Spec-spice-loop7	0	-140
Spec-spice-loop8	0	3
Spec-spice-loop9	0	3
Spec-spice-loop10	0	1
Whet-cycle4_1	0	0
Whet-cycle4_2	0	1
Whet-cycle4_4	0	2
Whet-cycle4_8	0	-2
Whet-loop1	0	3
Whet-loop2	1	2
Whet-loop3	0	2

Table 5.2: Optimal Solutions : Difference between Hamiltonian SIRA and SIRA

				Min R with limits			
Loop	$\|V\|$	$\|V_R\|$	MII	reach ≤ 16	time ≤ 5'	space $\leq$ 20MB	# int sol ≤ 3
Liv-loop23	20	19	8	16	14	14	14
Spec-tom-loop1	15	12	22	7	6	6	5
Spec-dod-loop1	13	12	22	9	6	6	7

Table 5.3: SIRA Solutions using Heuristics

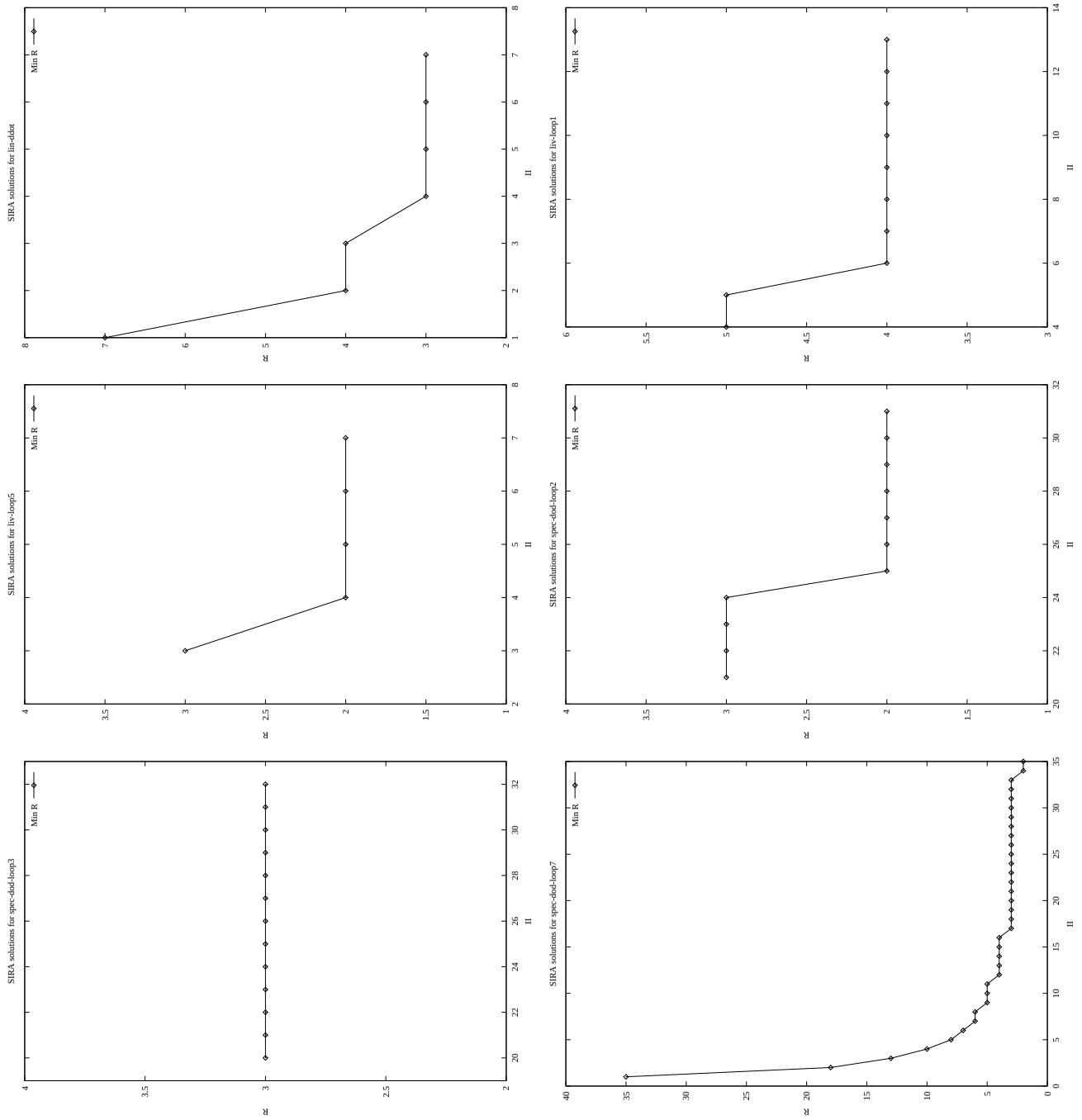


Figure 5.11: Detailed SIRA Solutions (Part 1)

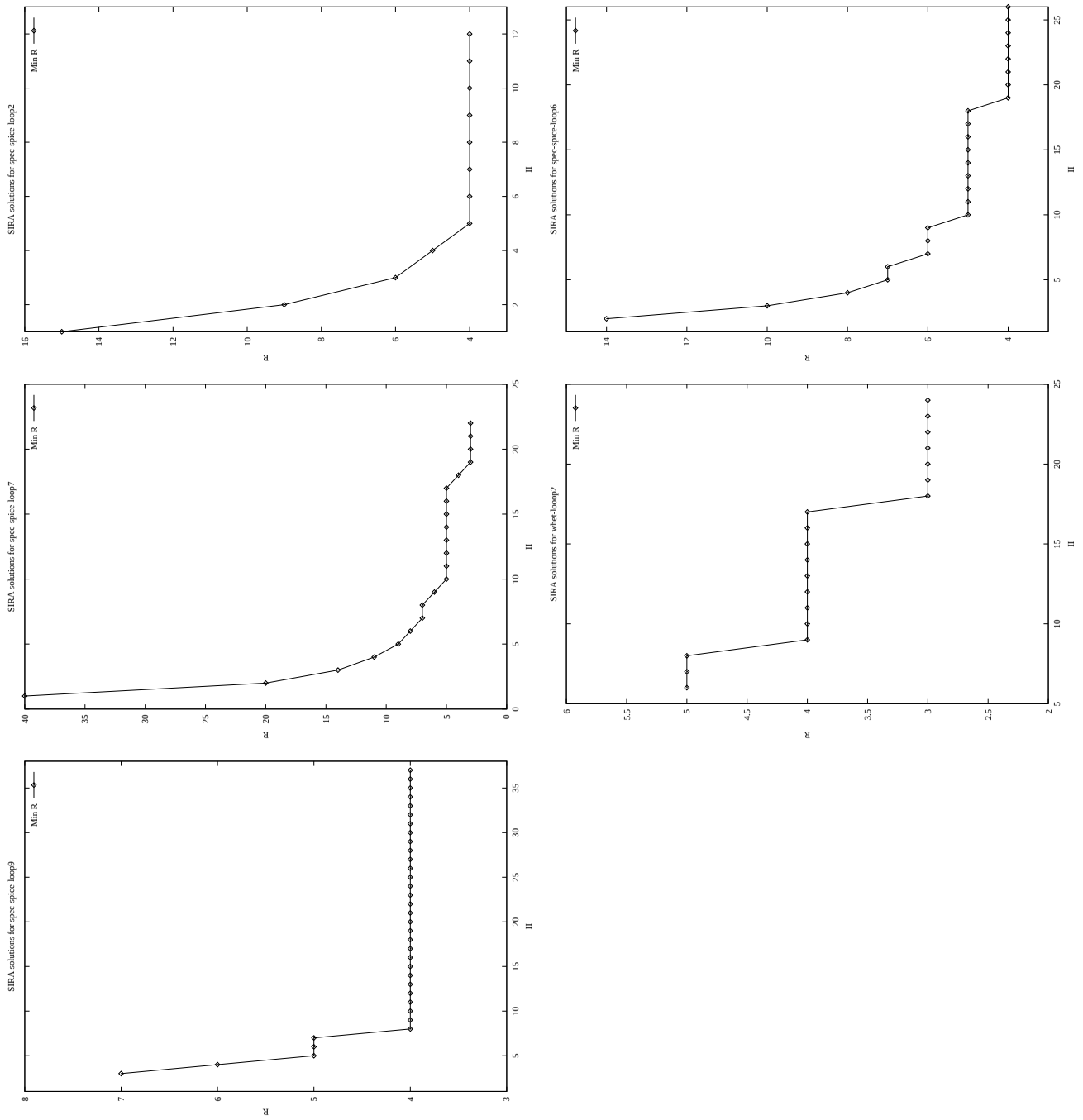


Figure 5.12: Detailed SIRA Solutions (Part 2)

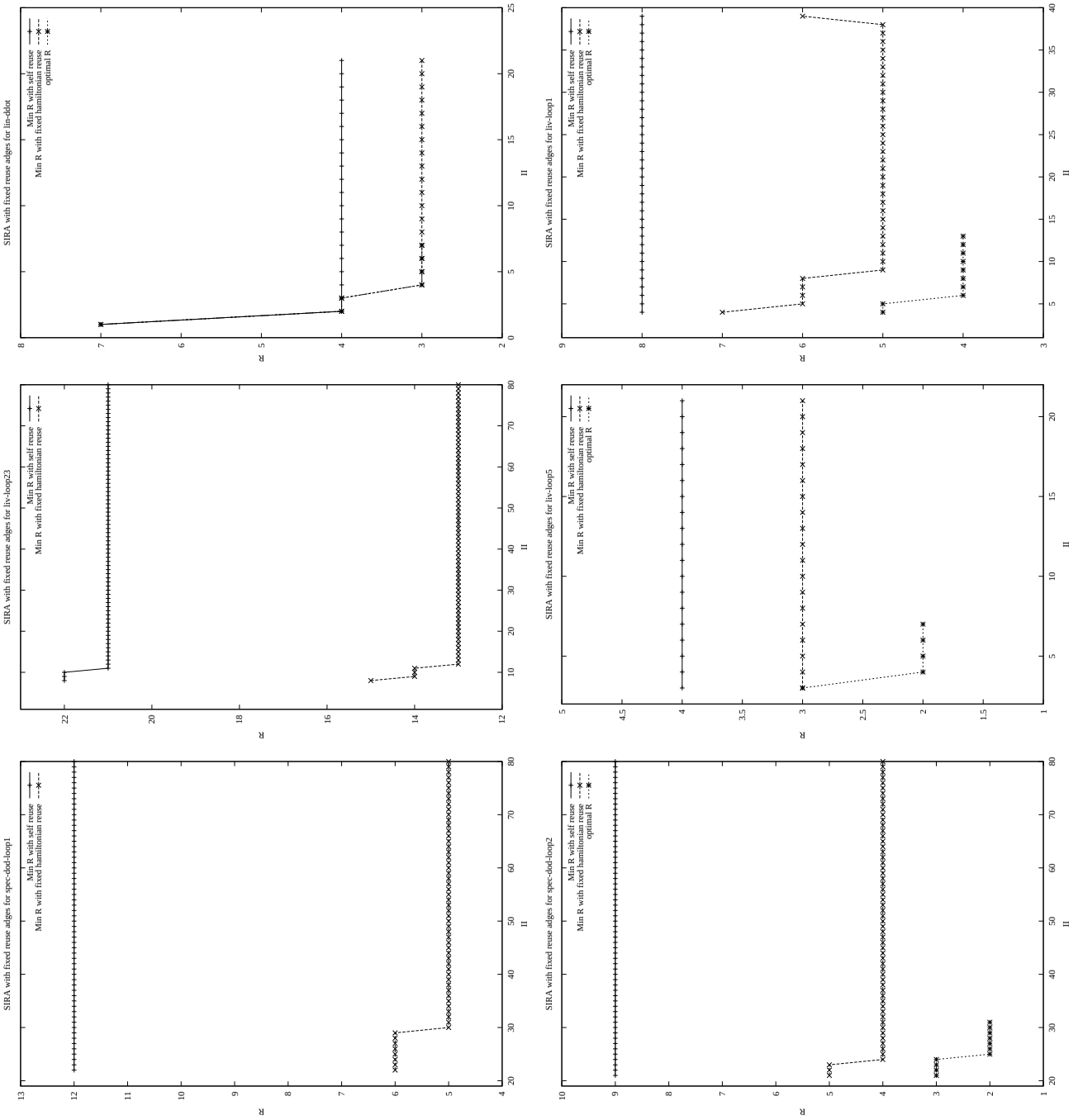


Figure 5.13: SIRA with Fixed Reuse Arcs (Part 1)

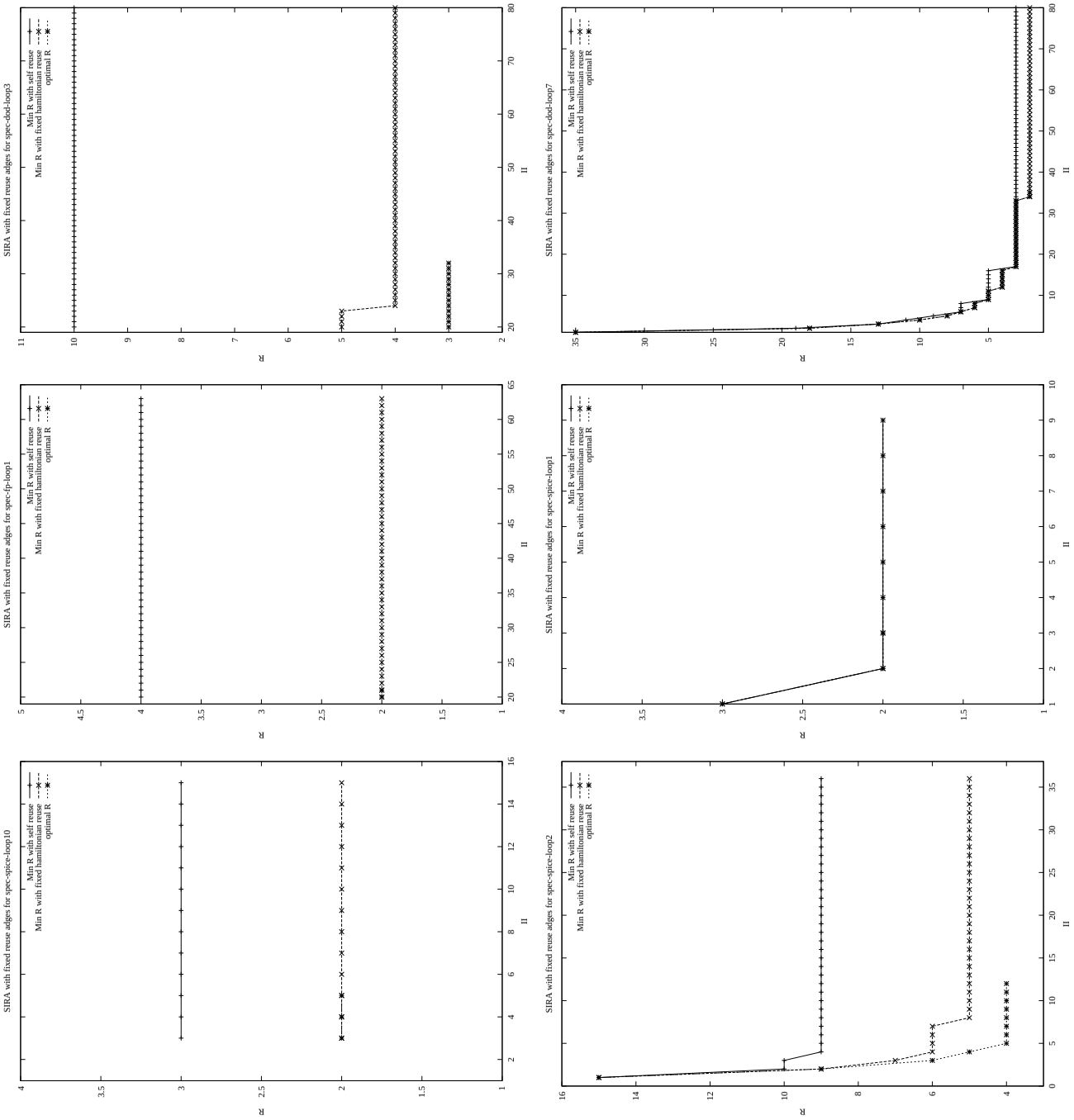


Figure 5.14: SIRA with Fixed Reuse Arcs (Part 2)

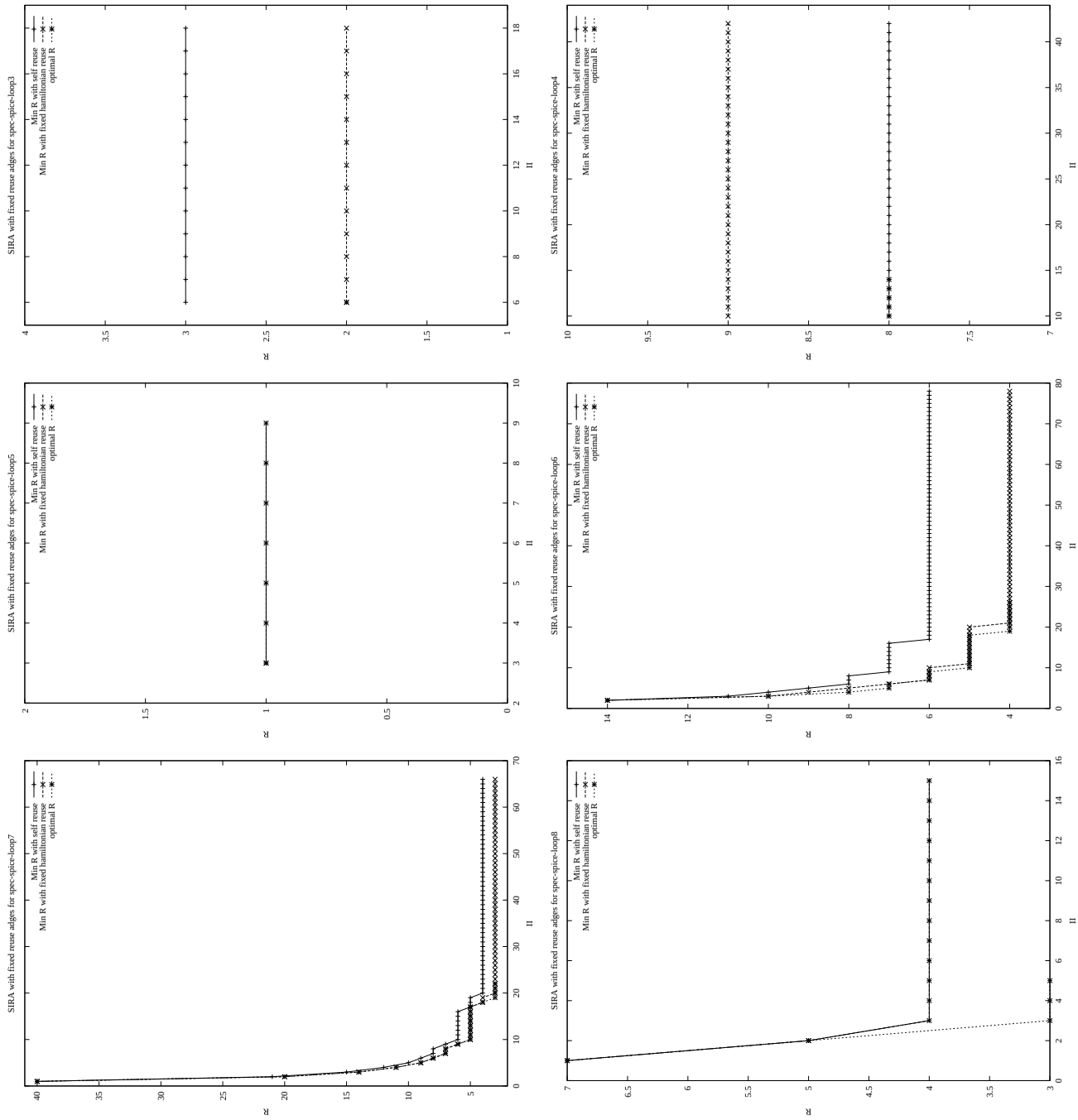


Figure 5.15: SIRA with Fixed Reuse Arcs (Part 3)

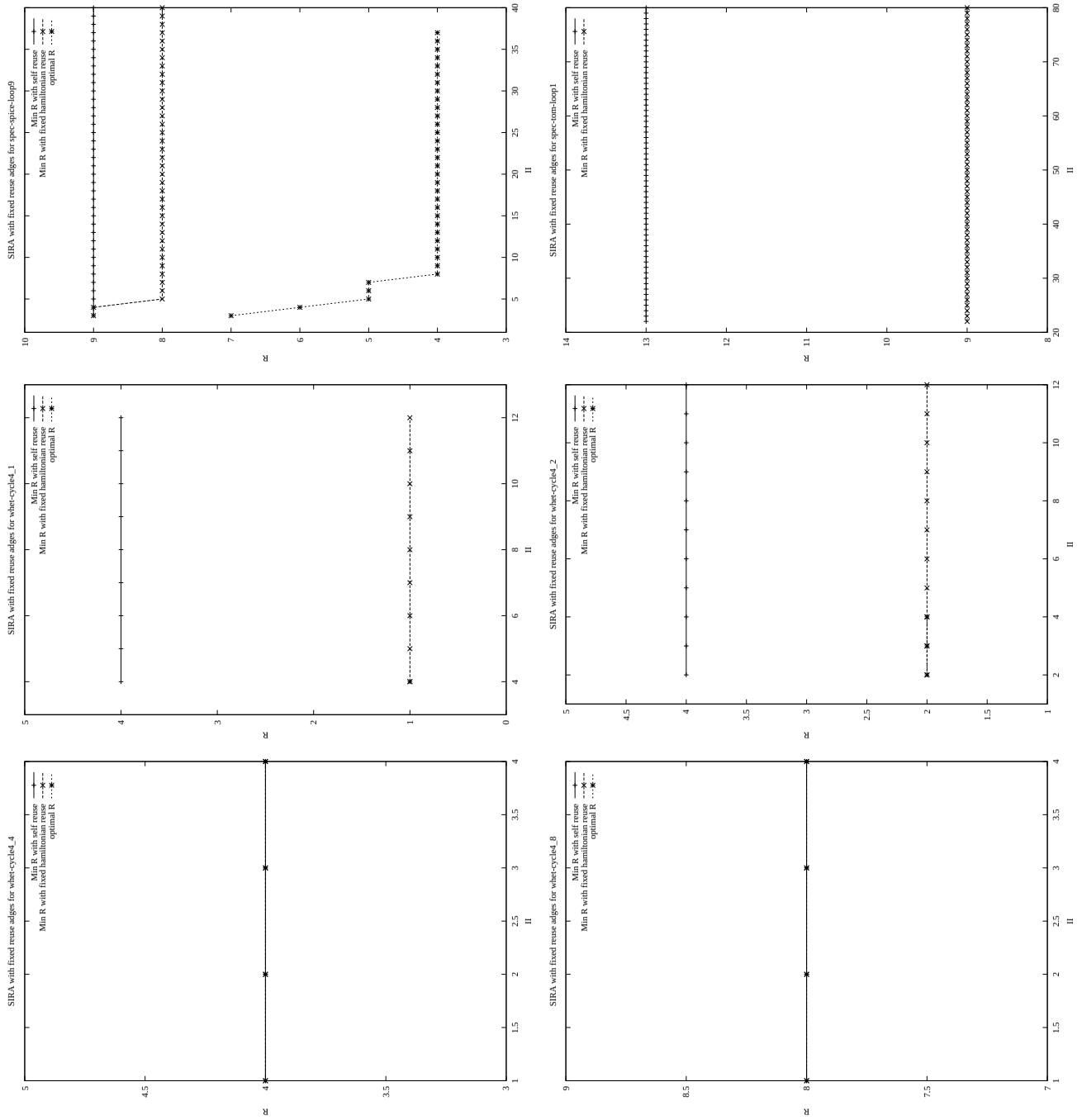


Figure 5.16: SIRA with Fixed Reuse Arcs (Part 4)

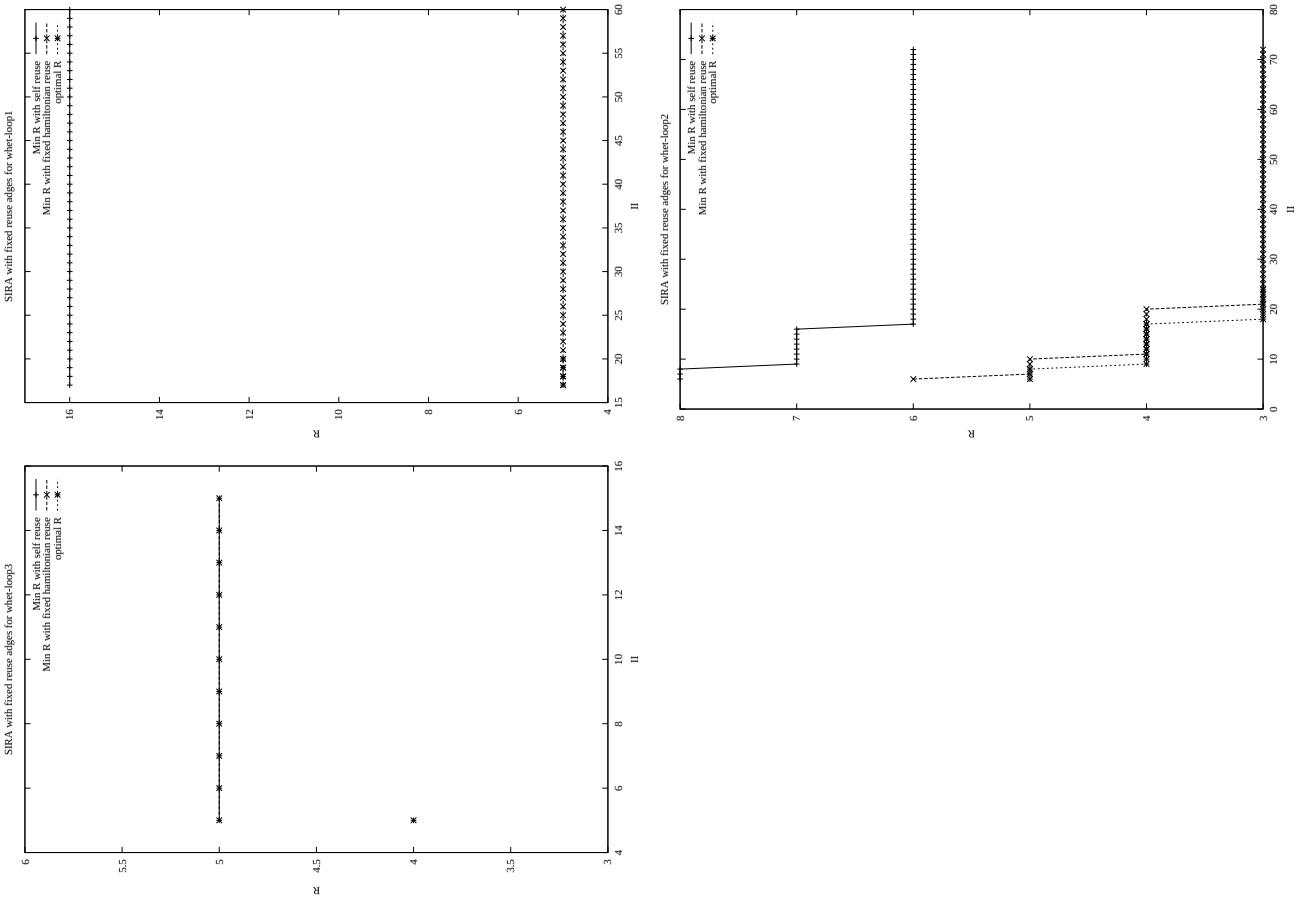


Figure 5.17: SIRA with Fixed Reuse Arcs (Part 5)

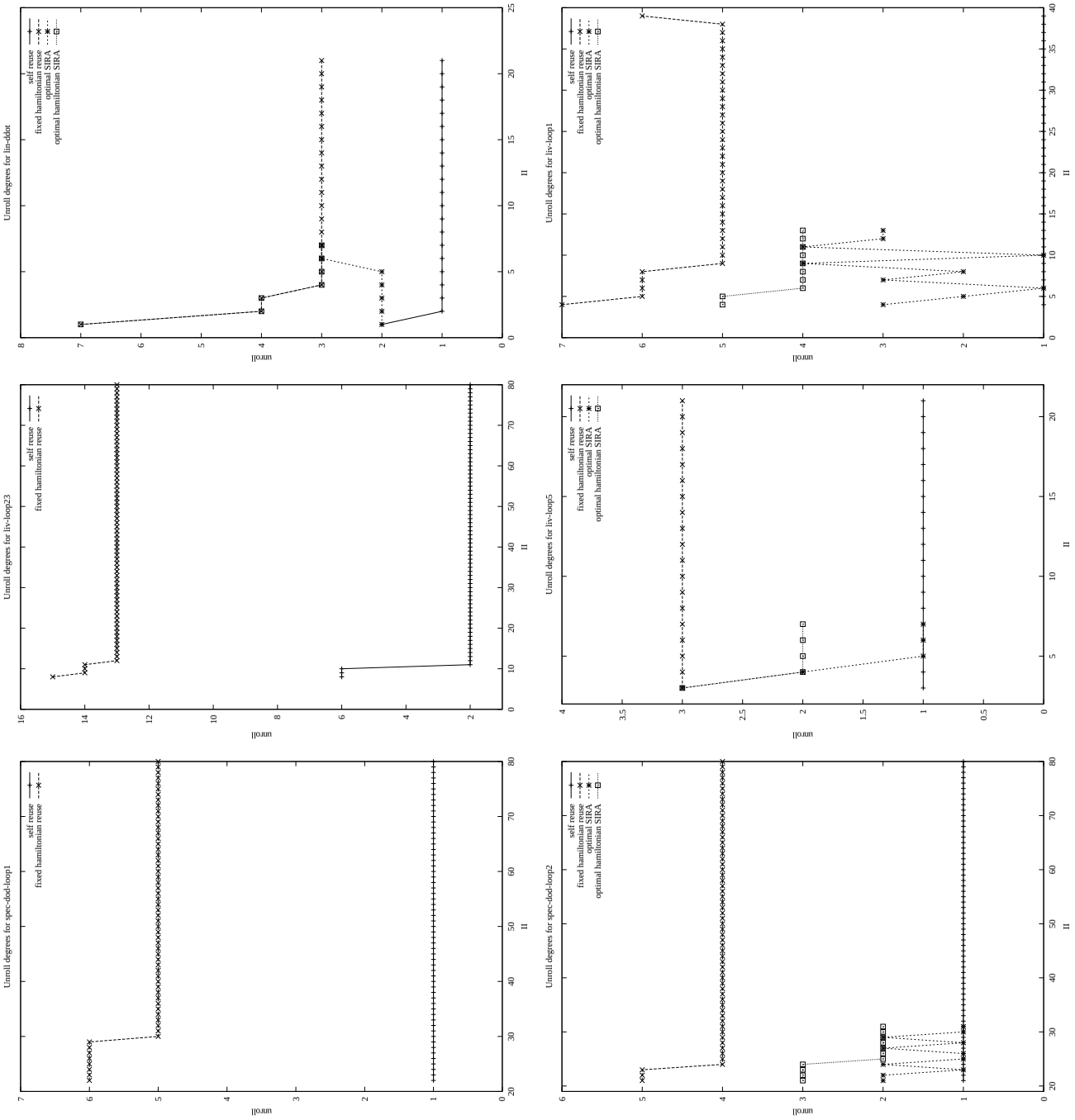


Figure 5.18: Unrolling Degrees (Part 1)

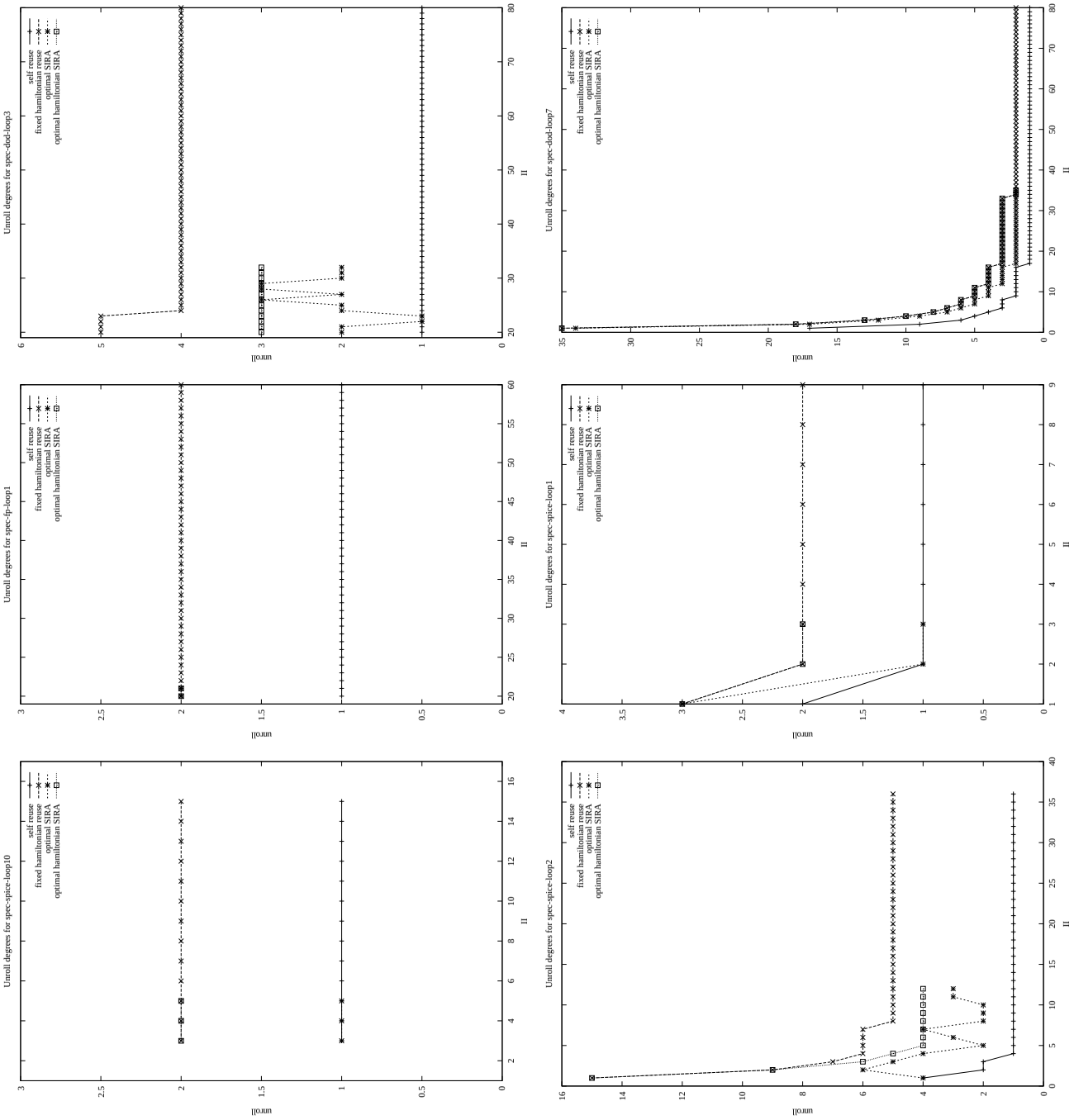


Figure 5.19: Unrolling Degrees (Part 2)

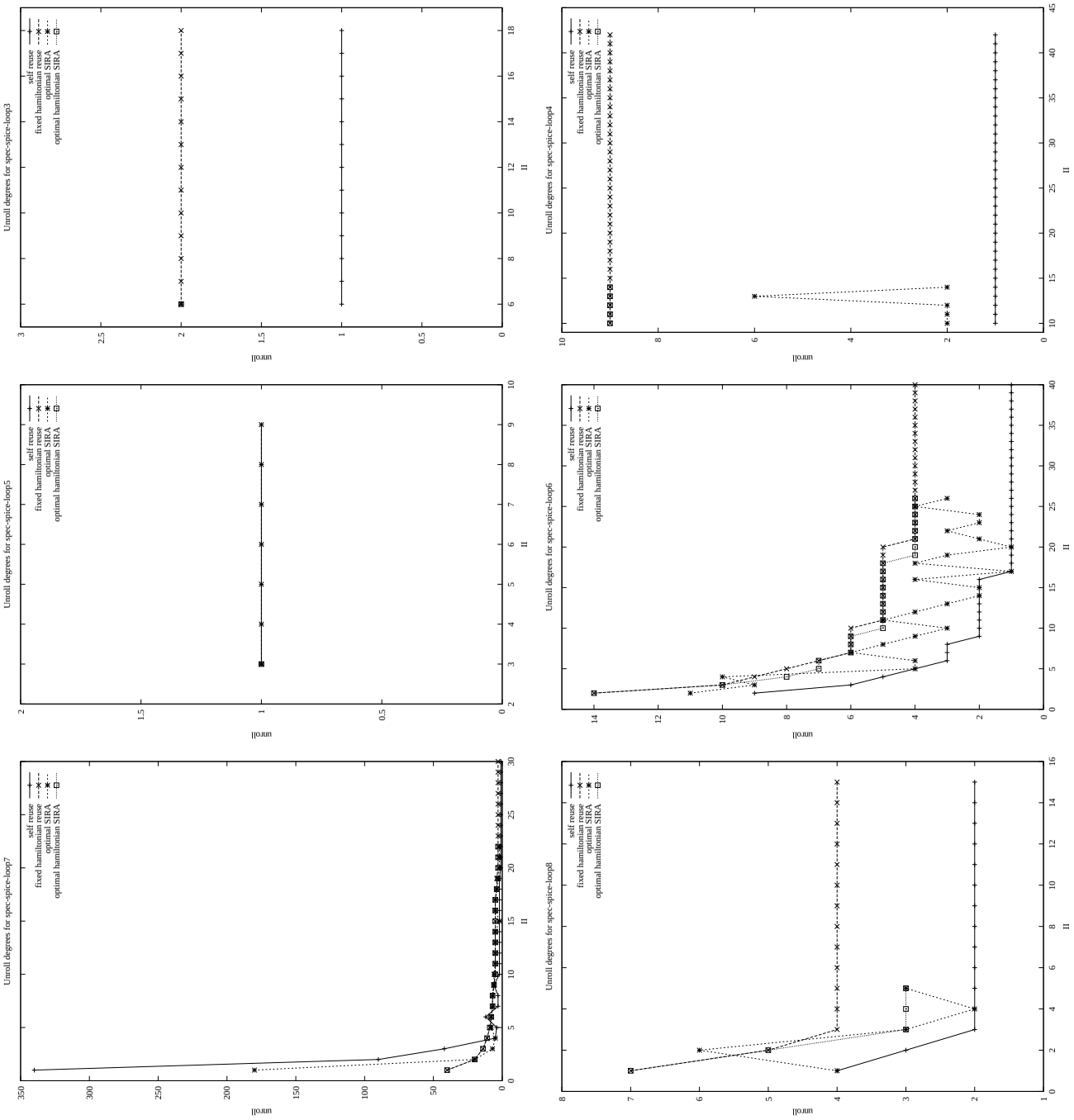


Figure 5.20: Unrolling Degrees (Part 3)

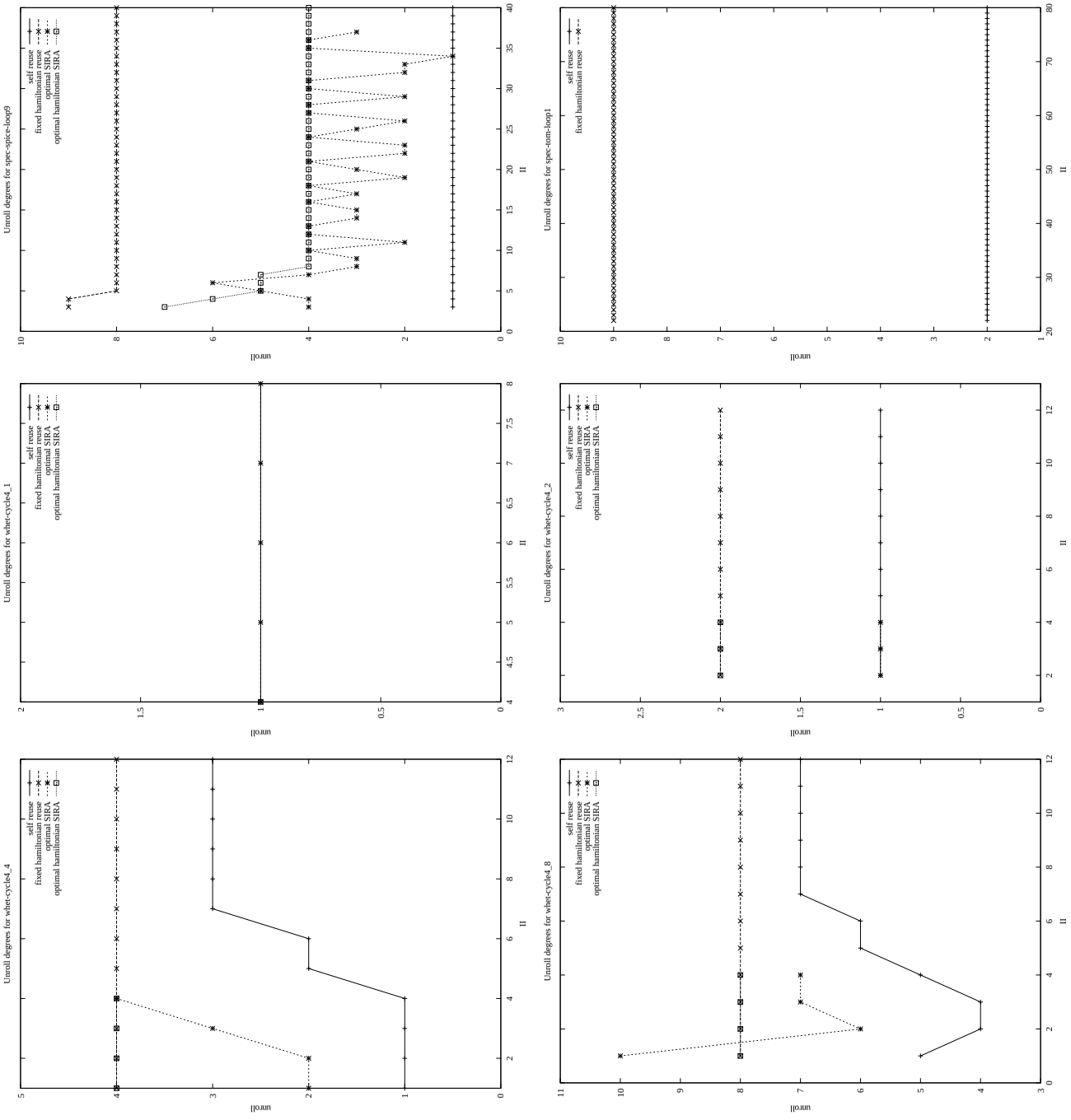


Figure 5.21: Unrolling Degrees (Part 4)

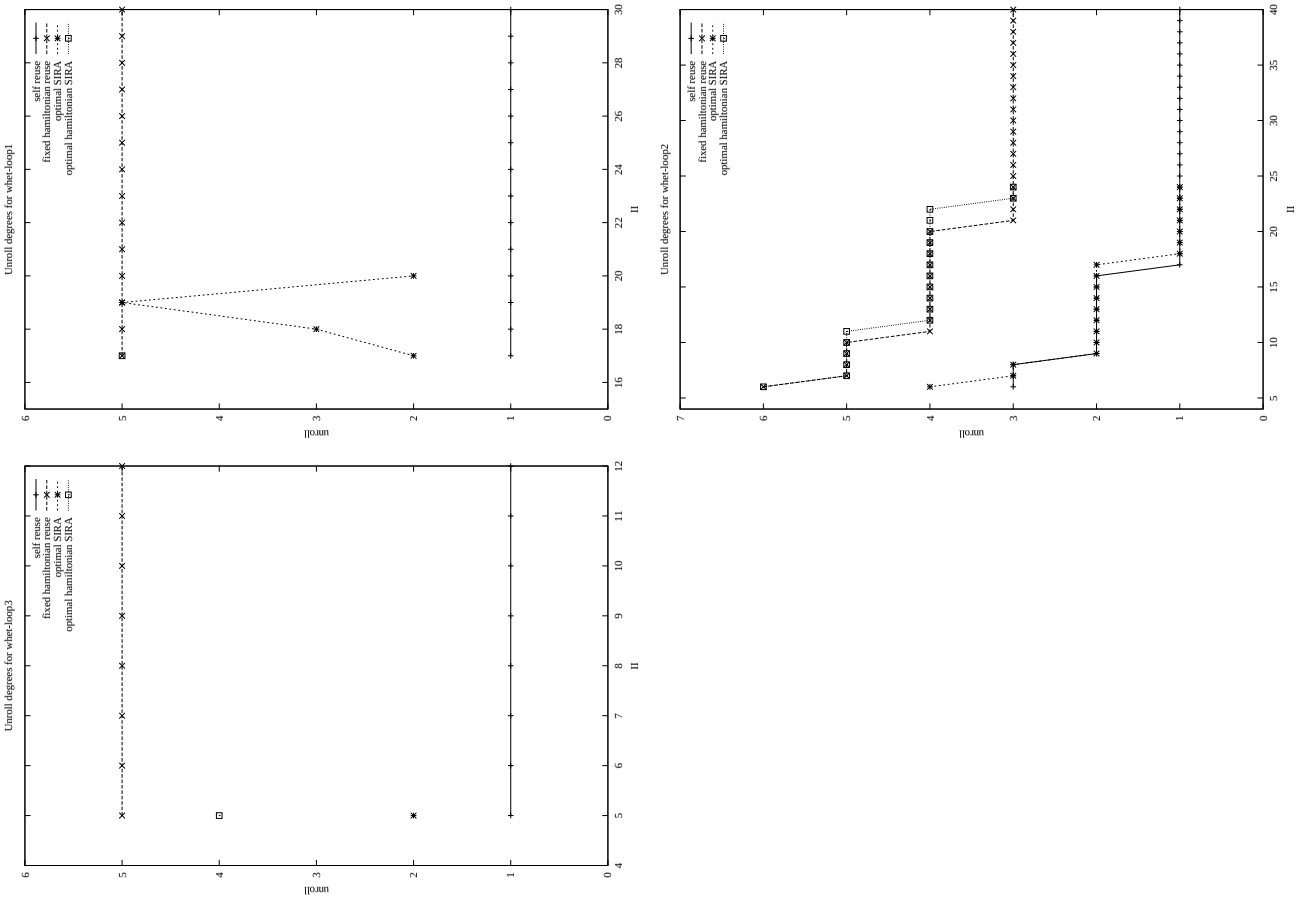


Figure 5.22: Unrolling Degrees (Part 5)

Chapter 6

Related Work in Loops

This chapter draws up a panorama on most important work in the field of register pressure in loops. Most of the techniques are based on SWP scheduling with limited number of registers.

6.1 Cyclic Register Saturation and Sufficiency

As far as we know, there is no study on CRS and CRF. The only work that may be considered as related to our study (up to our knowledge) was provided by Lilja and Bird in [LB94], and William Mangione-Smith *et al* in [MSAD92]. They build an approximated linear analytical model for the register requirement. They assume that a new register is allocated to each value at a constant rate (every h step). With this linear model, they are able to give a conservative approximation of upper and lower bounds for cyclic register requirement. Their approximations are not tight (exact) in the sense where they cannot guarantee the existence of a schedule that needs the computed register count.

6.2 Software Pipelining under Register Constraints

SWP with register constraints tries to ensure that the number of values simultaneously alive does not exceed the number of available registers, while guaranteeing the existence of a register allocation with the set of available registers.

Huff's Technique [Huf93] Huff [Huf93] was the first who proposed a SWP heuristics which tries to minimize values lifetimes, hoping that this would minimize the register requirement. It is based on defining for each statement an interval of possible issue times, called *slack*, depending on circuit dependences. Initially, slacks contain as soon and as late as possible issue times that are dynamically updated during scheduling. Some statements are scheduled early while others are delayed. Slack length defines a priority: the longer is a slack, the more freedom we have to schedule the statement, and the less is its priority. Backtracking is used to cancel computed issue slots if a statement cannot be scheduled within its slack.

Buffers Minimization [NG93, Nin93] Ning and Gao [NG93, Nin93] defined an approximation of register requirement called buffers. The difference between a buffer and a register is

that if two lifetime intervals do not interfere with each other, they can share a register but not a buffer. In fact, a buffer is a special register which passes the successive copies of the values produced from one SWP motif to successive ones [1]. The authors claim that buffer minimization is a polynomial problem, but their proof is not correct. Indeed, they use an approximation of buffers in order to prove that their constraints matrix is totally unimodular. However, their linear constraints compute an upper-bound of buffers, not the exact number. Nonetheless, buffers can be considered by our polynomial SIRA methodology when we fix self-reuse arcs.

Decomposed SWP [WKEE94, WKE95] Wang *et al* [WKEE94, WKE95] proposed a SWP technique that builds a kernel with a reduced register requirement. Their algorithm dynamically maintains a graph that reflects an approximation of register requirement during scheduling. Their model uses a similar formulation to SIRA (with reuse edges), but with restrictions. First, they assume that each value is consumed by only one operation and hence they did not investigate killing nodes. Second, they make reuse decisions according to anti-dependences present in the original code : if there is an anti-dependence between two statement in the original code, then they report this decision to scheduling constraints. They proved that when we fix row numbers (i.e. when the reservation table of the kernel is computed) and original anti-dependences (fixed register reuse), then finding columns numbers that minimize register requirement under a fixed II is a polynomial problem. In fact, it is a special case of Theorem 5.5 (Section 5.6).

RESIS [SC96] RESIS methodology [SC96] tries to minimize MAXLIVE in an existing kernel. Their algorithm has two main steps. First, they build a new DDG from an existing SWP motif. Variable lifetimes are shortened by reducing the iteration index of some statements. This is similar to defining column numbers via retiming as we do in Section 4.1.2. Second step tries to reduce MAXLIVE defined inside the kernel by computing row numbers so that interferences are reduced.

Hypernode Reduction and SWING Modulo Scheduling [LVA95, LGAV96, Llo96] HRMS for Hypernode Reduction Modulo Scheduling [LVA95] is a heuristics which constructs a SWP motif that shortens lifetimes while minimizing II at the same time. Before scheduling, operations are ordered so that only all direct predecessors of a node u or only all direct successors of u are scheduled before treating u. That is, authors avoid scheduling direct predecessors and direct successors before scheduling u itself. According to which node has been scheduled first, their direct predecessors or successors are scheduled as soon or as late as possible. However, this technique does not distinguish the statements : those belonging to critical circuits are more critical than others.

SWING [LGAV96, Llo96] overcomes HRMS drawback by taking into account latencies. Statements producing values are placed near to their consumers in order to shorten lifetimes. Full priority is given to statements belonging to critical circuits.

[1] Buffers are similar to circuit registers with maximal register sharing in Leiserson and Saxe terminology [LS91].

Universal Occupancy Vector [SCFS98] Strout *et al* [SCFS98] give a theoretical formulation of the relationship between storage (memory) requirement and parallelism in a loop nest. When an iteration $\vec{i}$[2] stores a value in the same location used by another iteration $\vec{j}$, this creates an output dependence between these two iterations with distance $\vec{j} - \vec{i}$. Furthermore, if an iteration $\vec{k}$ reads the value defined by $\vec{i}$, this creates an anti-dependence with distance $\vec{k} - \vec{i}$. These distances are called Universal Occupancy Vectors (UOV). The introduced false dependences because of storage limitations create new circuits that limit the throughput MII. So the problem is to find these UOVs. The authors show that determining if a vector is a UOV is NP-complete and propose an algorithmic approximation to find a good one.

Our reuse relation studied in previous chapter may be considered as a variant of UOV since registers are indeed a memory. However, UOV defines reuse patterns between iterations and not statements. Hence, it cannot be used for register allocation because it does not model precisely reuse relationship between statements. Furthermore, registers are slightly different than classical memory cells since they are accessed directly (without addressing) and involve loop unrolling to be allocated.

Recently, Thies *et al* [TVSA01] presented an application of UOV vectors for affine scheduling of loop nests. They presented an elegant unified framework to determine a good storage mapping for a given schedule, a good schedule for a given storage mapping, and good storage mapping that is valid for all legal affine schedules. Their technique has a direct application in the context of array expansion, where the cost of adding one dimension to an array may give more freedom for parallelism (removal of false dependences). Our SIRA reuse model can be considered as a specialization of this theoretical framework, since we only consider registers as a storage mapping for innermost loops. However, reuse graphs are especially thought up to be directly applied to cyclic register allocation in ILP codes. Loop unrolling and hamiltonian reuse circuits are modeled in a better and simpler way with reuse graphs. Furthermore, our reuse model enables us to prove that finding the best reuse distances, with fixed reuse arcs, is a polynomial problem.

Integer Programming Techniques Integer linear programming to build a SWP schedule under register constraints was first introduced by Altman [Alt95, GAG94]. However, he did not exactly express the register requirement, but an approximation based on the buffers. Thus, it cannot be considered as an exact formulation of register need.

Sawaya [ES96a, ES96b, Saw97] wrote an integer programming model which reduced the exact register requirement. The complexity of his model was $\mathcal{O}(|V| \times \lambda_{max} h)$ variables and $\mathcal{O}(|E| + |V| \times \lambda_{max} h)$ constraints. Coefficients inside constraints matrix are upper-bounded by $L_{max} \times \lambda_{max}$.

Another formulation was given in [EDA96] with $\mathcal{O}(|V| \times h)$ variables and $\mathcal{O}(|E| + |V| \times h)$ constraints, in which the coefficients inside constraints matrix are upper-bounded by $h \times \lambda_{max}$. However, this model needs a fixed reservation table : the row numbers must be computed and fixed in a first step so as to satisfy resource constraints. Then, it tries to find column numbers that minimize MAXLIVE. A similar formulation to Eichenberger's intLP system was recently given by Huard in [Hua01]. Indeed, the size of his constraints matrix has the same complexity

[2]Recall that iterations in multidimensional loops are vectors.

than Eichenberger's method. However, Huard proved that this problem is NP-complete in the strong sense (minimizing MAXLIVE under fixed row numbers and initiation interval).

Recently, Fimmel *et al* have written in [FM01] an exact formulation of software pipelining under register and resource constraints. Since they compute the number of values simultaneously alive at each time step within $[0, h[$, their intLP system generates an equivalent number of variables and constraints as Sawaya's method. However, as in our model, they assume writing delays (offsets) such that a register does not have to be occupied before the operation result is available. Furthermore, they remarked that when sharing of registers is disabled (as our self-reuse strategy), their intLP system is considerably simplified. Indeed, we proved in the last chapter that this problem is polynomial and can be formulated with a totally unimodular constraints matrix. Their constraints matrix wasn't proven so.

All the above intLP techniques suffer from their model size. Since they introduce h in their size complexity, constraints matrix growth depends on specified latencies (input data) and how nodes are connected (structure of the DDG). This is because they define an integer variable for each clock cycle within the interval $[0, II[$ that computes values simultaneously alive. Our modeling is $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E| + |V|^2)$ constraints while coefficients are bounded by $\pm L_{max} \times \lambda_{max}$. This is because we compute MAXLIVE by using circular intervals, i.e. only during dates when a value is defined or killed. Hence, the number of variables and constraints in our intLP model depends only on the *size* of input DDG.

If we succeed in finding a SWP schedule that does not require more than R registers, register allocation with R available registers can be performed. Some work in this field is explained below.

6.3 Register Allocation of Software Pipelined Loops

Hendren's Approach [HGAM92, H$^+$92] Laurie Hendren *et al* [HGAM92, H$^+$92] proposed a heuristics for cyclic register allocation based on an empirical remark: in almost all cases, a cyclic graph is R or $(R+1)$ -colorable (R is the maximal number of values simultaneously alive). Thei heuristics proceeds by first trying to color the intervals which cross the motif barrier. The intervals inside the motif itself are acyclic and hence can be easily colored. If there are not enough registers, spill code is introduced. Cyclic life intervals with multiple turns around the motif may contain several colors which correspond to the multiple copies of values: a first approach introduces some shift operations to move these copies from one register to another. Introducing these extra operations increase the initiation interval of the motif. Another approach consists in unrolling the loop to exhibit the different copies and to allocate each copy to a different register.

Rotating Register Files [RLTS92] Rau [RLTS92] proposed a method for a cyclic register allocation if a rotating register file (RRF) is present. After determining the SWP motif, circular lifetime intervals are completely defined. If the underlying hardware does not implement a RRF, we must unroll the loop and rename the copies of values in order to avoid conflicts. In the presence of a RRF, we only consider the motif without unrolling for cyclic register alloca-

tion. The problem becomes to fit all the circular intervals into a cylinder where its axis is the time (in terms of clock cycles) while minimizing its circumference as shown in Section 2.4.1. Experimental results show that in 80% of the cases, the gain in terms of required registers on a RRF is not significant compared to loop unrolling, but the code is more compact.

Software Simulation of RRF [DGS92] Duesterwald *et al* [DGS92] introduced the concept of *register pipeline* to improve the register reuse between iterations. It is a set of registers allocated to lifetimes intervals without considering copies of values. It is indeed a sort of software simulation of a rotating register file. Their approach consists of a variant of graph coloring with multiple colors, since multiple physical registers may be assigned to the same value to hold all its copies. In the presence of a RRF, the code is easily generated without loop unrolling. Otherwise, they introduce move operations to simulate a RRF, which may increase the II and may need rescheduling the code if these move operations do not fit into the kernel.

Meeting Graphs [ELM95, Lel96, ELM97, dWELM99] A complete theoretical framework on cyclic register allocation was made by Eisenbeis and Lelait [ELM95, Lel96, ELM97, dWELM99]. They introduce the meeting graph structure defined in Section 2.4. They show how to always find a cyclic register allocation with R registers if we sufficiently unroll the already scheduled SWP motif. They proceed by decomposing the meeting graph into elementary circuits, in which each circuit correspond to a reuse pattern. The drawback was that the unrolling factor depended on the circuit decomposition, and it was difficult to succeed in finding a circuit decomposition with a minimized unrolling factor. However, in the presence of a rotating register file, a cyclic register allocation may be done without unrolling [Lel96] if the meeting graph contained an hamiltonian circuit. If no such circuit is present, they need a rotating register file with $R+1$ instead of R registers to build a cyclic register allocation.

6.4 Conclusion

This chapter presents most of related work in the field of register pressure in modulo scheduled loops. While no study has been done in cyclic register saturation and sufficiency (as far as we know), many strategies rely on scheduling with a limited number of registers. Register Allocation of such pipelined loops with R values simultaneously alive needs R registers if loop unrolling is applied, or at most $R+1$ in the presence of a RRF (without loop unrolling). Almost all techniques described in this chapter do not consider neither multiple register types nor explicit delays in reading from and writing into registers, while our model do.

Our approach is different since it takes into account register constraints prior to scheduling. Two main strategies have been explored. First, the CRS and CRF analysis enables us to guarantee the existence of at least one valid SWP schedule under a fixed number of registers with an optimized critical circuit. Then, we can apply scheduling and register allocation in any order we want. The second strategy (SIRA) consists in applying cyclic register allocation, prior to scheduling, directly into the DDG while minimizing the critical circuit.

Chapter 7

Conclusion

This work addresses the problem of register pressure in simple loops independently from the scheduling process. Our aim is to give a higher priority to the register allocation in order to exploit at the best the registers available in the target processor. The register pressure is defined as a triplet consisting in the saturation, the sufficiency and the number of available registers. Given these three factors, we have enough information to guarantee before the scheduling that the loop performance would or not suffer from the registers constraints. Two approaches have been studied : adding serial arcs before the scheduling to limit the cyclic register requirement, or allocating the registers before the scheduling. In both cases, we minimize the increase of the critical cycle to reduce the ILP loss.

Our architecture model has regular register sets. However, in some architectures, the register types are not canonical, i.e. some operations can have the choice of writing into more than one register set. We can for instance decide at the beginning for the register type that a value produces, but this would restrict the ability of using more available registers since we cannot know at advance which register type is suitable for a value. Another approach can be an iterative strategy : we can use a heuristic to decide at the beginning in which register type resides a value, and try to fit the registers constraints for all the register types for the fixed decision. If not, iterate over another decision. However, the number of choices may be combinatory in function of the number of register types. We can circumvent this problem by considering virtual register types as described in [ZW01] which tried to extend [CER99]. A virtual register type is a combination of the original types. This creates new canonical virtual types where each new type is composed of a union of some of the original types. This allows us to use our loop/architecture model. But at the end, we must come back to the original register types by doing a register assignment phase, i.e. to decide in which type of registers resides a value. Unfortunately, contrary to what has been stated but not proven in [ZW01], the existence of a valid schedule under the register constraints with the virtual types does not guarantee the existence of a valid register assignment. This is because a value lifetime interval may need to change a register type at a certain point of time in order to do not exceed MAXLIVE. Of course, this can be done by inserting shift operations, but this is another issue.

Some studies [LMEG96, Jan01] claimed that inserting spill code in modulo scheduled loops is better than increasing the II. We do not adhere to this thesis at all. The reason why these authors made this claim is that, first, they assumed static memory operation latencies and they found that the scheduler succeeded in most case to find a free slot for the inserted spill code.

Second, they made an amalgam between the static loop performance defined by the II and the real (dynamic) one. The memory access operations have unforeseeable effects and can throw into disorder the scheduling process. Of course, we can be optimist and assume that the spilled values can reside in the cache. We do not prefer such assumption because any misprediction yields to a cache miss which results in a deep performance loss: in a superscalar processor, the fluidity of the dynamic execution of the pipelined loop is broken since the long miss latency scrambles the static schedule; also, a VLIW machine completely stalls. We prefer keep the dynamic execution under a static control when possible.

Our work does not address the case of loops with control dependencies. The reason is that the presence of tests inside the loop body prevents us from getting accurate flow dependencies information, and hence cyclic live ranges cannot be determined statically.

A future work consists of studying and extending our approaches to multi-dimensional modulo scheduling [Ram94, GQD94]. This will certainly provide the solution of the optimal register allocation in the case of a loop nest. A second perspective is to write an exact formulation of the optimal spilling problem for modulo scheduled loops under a fixed execution rate. Third, we can extend the reuse graphs in order to take into account the cache lines instead of the registers. The aim is to provide some compilation techniques for the software managed caches, where the compiler has the control on the replacement policy. The reuse arcs would express the fact that two memory operations reuse the same cache location. The problem would be for instance to prevent the loop from accessing more cache lines than the cache capacity, or to decide which memory line should reside in the cache. Note that the cache lines can also be replaced by the memory cells, and hence another perspective is to study some new memory management techniques (used for out-of-core computation, data layout optimization, etc.). In this case, a reuse arc would express the fact that two virtual memory addresses share the same physical memory location. This perspective is a continuation to some existing work about the tradeoff between the parallelism and the storage requirement in a loop nest [SCFS98, TVSA01].

Appendix A

Proof of Theorem 5.5

In this section, we prove that the constraint matrix of the following integer program is totally unimodular:

$$\begin{array}{ll} \text{Minimize} & \sum_{u \in V_{R,t}} \mu'_u \\ \\ \text{Subject to:} & \\ \mu'_u + \sigma_v - \sigma_{k_{u^t}} \geq -\delta_w(v) & \forall (k_{u^t}, v) \in E_r \\ \sigma_v - \sigma_u \geq \delta(e) - h\lambda(e) & \forall e = (u,v) \in (E_{\rightarrow r} - E_r) \end{array} \tag{A.1}$$

with $\mu'_u = h \times \mu^t_{u,v}$ where the reuse arcs are fixed. Let begin by the definition of the incidence matrix of a directed graph.

Definition A.1 (Incidence Matrix) *Let be $D = (N, A)$ a directed graph with N the set of nodes and A the set of arcs. The incidence matrix U of D is a $\{0, \pm 1\}$-matrix, where the rows are indexed by the arcs and the columns indexed by the nodes, such that:*

$$U(e,n) = \begin{cases} +1 & \textit{if } e \textit{ enters } n \\ -1 & \textit{if } e \textit{ leaves } n \\ 0 & \textit{otherwise} \end{cases}$$

The incidence matrix of any directed graph is totally unimodular [Sch87].

System A.1 is detailed in Figure A.1. The first $|V_R|$ variables are the μ' distancies (one for each value $u \in V_R$), and the remaining $|V_{\rightarrow r}|$ variables are the scheduling variables σ_u (one for each $u \in V_{\rightarrow r}$). We decompose the coefficients in the constraint matrix into three main submatrix or "sectors":

1. the first sector corresponds to the coefficients of the μ' variables for the $|E_r| = |V_R|$ first linear constraints. Since the reuse relation is a bijection, this submatrix is square. Furthermore, there is one and only one $+1$ in each row and each column in this submatrix. So, we can make a permutation of the columns in this submatrix[1] to get a square identity submatrix with the dimension $|V_R| \times |V_R|$;
2. the second sector corresponds to the coefficients of the μ' variables for the remaining $|E_{\rightarrow r}| - |E_r|$ linear constraints. This submatrix is totally null;

[1] It corresponds to a simple variable renaming.

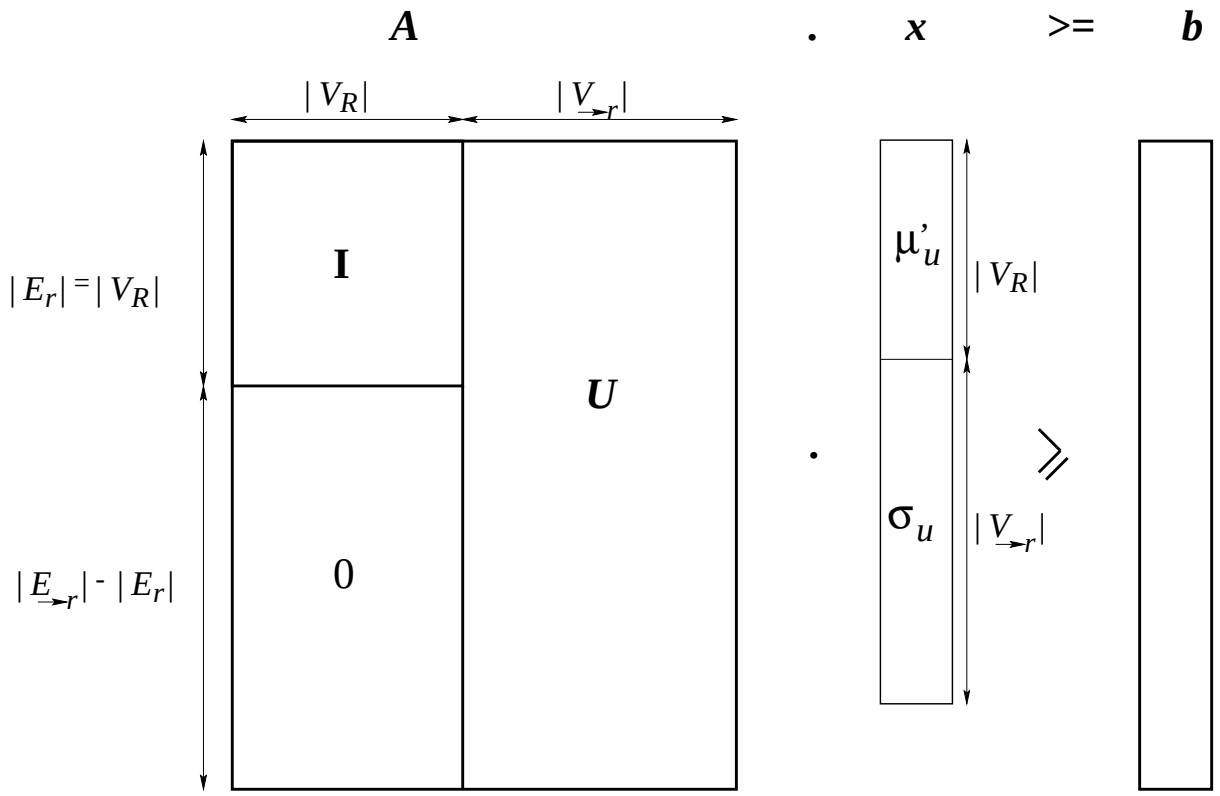


Figure A.1: The Constraint Matrix of System A.1

3. the third sector corresponds to the coefficients of the σ_u variables for all the constraints. This submatrix is the incidence matrix of the DDG $G_{\rightarrow r} = (V_{\rightarrow r}, E_{\rightarrow r}, \delta_{\rightarrow r}, \lambda_{\rightarrow r})$.

Now, we are ready to investigate all the case of square submatrix of A to see that their determinant is either null or ± 1. There are exactly seven cases, numbered from 1 to 7 as shown in Figure A.2. Let be A' an arbitrary an n-square submatrix of A.

Case 1 if A' belongs totally to the first sector, then it is a square submatrix of an identity and hence

$$det(A') \in \{0, 1\}$$

Case 2 $det(A') = 0$, trivial !

Case 3 A' contains a part of rows/columns of an identity and a part of rows/columns of an incidence matrix, but not a part of the null submatrix. If there is at least one null column in the identity part, then the determinant is null. Otherwise, all the columns from the identity part contain at least one $+1$. We can permute the columns of the identity part to get the matrix shown in Figure A.3.(a). The determinant of A' is then equal to the determinant of U', a square submatrix of the incidence matrix. Hence :

$$det(A') = det(U') \in \{0, 1, -1\}$$

Case 4 $det(A') = 0$, trivial !

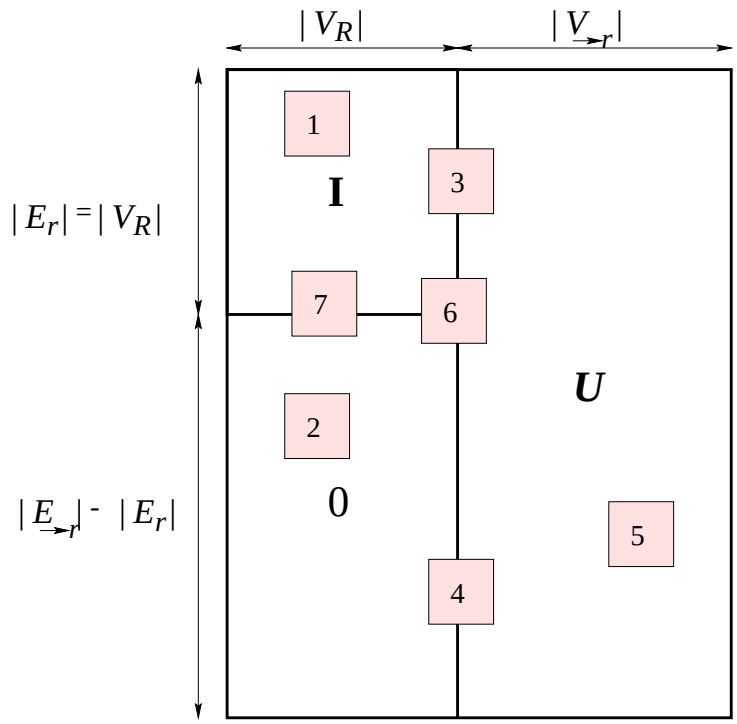


Figure A.2: All Possible Square Submatrix

Case 5 A' is a square submatrix of a totally unimodular matrix (incidence matrix). So

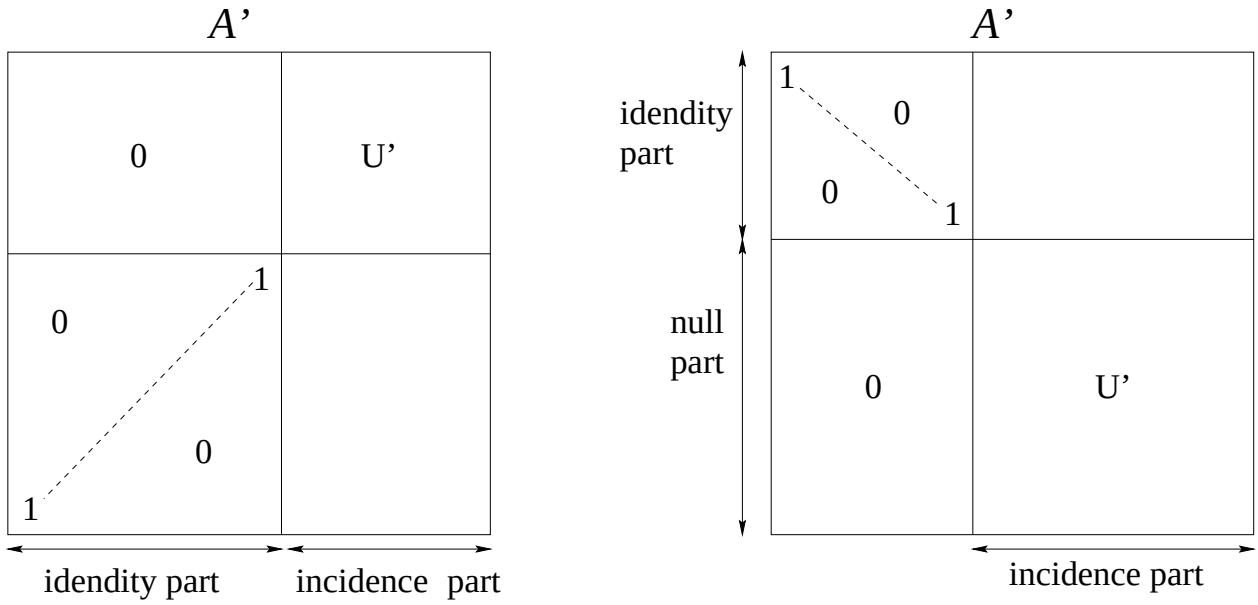
$$det(A') \in \{0, 1, -1\}$$

Case 6 A' is an intersection of three sectors. If there is at least one null column in the identity part, then the determinant is null. Otherwise, all the columns from the identity part contain at least one $+1$. We can permute the rows and columns of the identity part to get the matrix shown in FigureA.3.(b). The determinant of A' is then equal to the determinant of U', a square submatrix of the incidence matrix. Hence:

$$det(A') = det(U') \in \{0, 1, -1\}$$

Case 7 $det(A') = 0$, trivial !

Finally, note that any bounding information on the integer variables do not alter the unimodularity of the constraints matrix. This is because bounding constraints only juxtapose (above or below) identity matrix to the original constraint matrix.



(a) Case 3

(b) Case 6

Figure A.3: Totally Unimodular Square Submatrix

Index

Bibliography

[AJLA95] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[Alt95] Eric Altman. *Optimal Software Pipelining with Functional Units and Registers.* PhD thesis, McGill University, Montreal, October 1995.

[Ber77] C. Berge. *Graphes et Hypergraphes.* Dunod, Paris, 1977.

[BG96] Rastislav Bodik and Rajiv Gupta. Array Data Flow Analysis for Load-Store Optimizations in Fine-Grain Architectures. *International Journal of Parallel Programming*, 24(6):481–512, December 1996.

[BS76] John Bruno and Ravi Sethi. Code Generation for a One-Register Machine. *Journal of the ACM*, 23(3):502–510, July 1976.

[Car91] M. Carlisle. *On Local Register Allocation.* PhD thesis, University of Delaware, 1991.

[CCK90] David Callahan, Steve Carr, and Ken Kennedy. Improving Register Allocation for Subscripted Variables. *ACM SIG-PLAN Notices*, 25(6):53–65, June 1990.

[CER99] Zbigniew Chamski, Christine Eisenbeis, and Erven Rohou. Flexible Issue Slot Assignment for VLIW Architectures. Research Report RR-3784, INRIA, October 1999. http://www.inria.fr/rrrt/index.en.html.

[CPL93] CPLEX Optimization, Inc., Incline Village, Nevada. *Using the CPLEX Callable Library and CPLEX Mixed Integer Library*, 1993.

[DET00] Min Dai, Christine Eiseinebis, and Sid-Ahmed-Ali Touati. Load-Store Optimization For Software Pipelining. *Computer Architecture News*, 28(1), March 2000.

[DGS92] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Register Pipelining: An Integrated Approach to Register Allocation for Scalar and Subscripted Variables. In *Compiler Construction, 4th International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 192–206. Springer, October 1992.

[DGS93] E. Duesterwald, R. Gupta, and M.L. Soffa. A Practical Data Flow Framework for Array Reference Analysis and its Application in Optimizations. *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–77, June 1993.

[DH00] Alain Darte and Guillaume Huard. Loop Shifting for Loop Compaction. *International Journal of Parallel Programming*, 28(5):499–??, 2000.

[DHB89] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped Loop Support in the Cydra 5. In *Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, New York, April 1989. ACM Press.

[DT93] James C. Dehnert and Ross A. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, 7(1–2):181–227, May 1993.

[dWELM99] Dominique de Werra, Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. On a Graph-Theoretical Model for Cyclic Register Allocation. *Discrete Applied Mathematics*, 93(2-3):191–203, July 1999.

[EDA96] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Minimizing Register Requirements of a Modulo Schedule via Optimum Stage Scheduling. *International Journal of Parallel Programming*, 24(2):103–132, April 1996.

[EGS95] Christine Eisenbeis, Franco Gasperoni, and Uwe Schwiegelshohn. Allocating Registers in Multiple Instruction-Issuing Processors. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT'95*, pages 290–293. ACM Press, June 27–29, 1995.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, April 1972.

[ELM95] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The Meeting Graph: A New Model for Loop Cyclic Register Allocation. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 264–267, Limassol, Cyprus, June 1995. ACM Press.

[ELM97] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. Circular-arc Graph Coloring and Unrolling. In U. Faigle and C. Hoede, editor, *Proceedings of the 5^{th} Twente Workshop on Graphs and Combinatorial Optimization*, pages 71–74, Twente, Netherlands, May 1997. Universiteit Twente.

[ES96a] Christine Eisenbeis and Antoine Sawaya. Optimal Loop Parallelization under Register Constraints. In *Sixth Workshop on Compilers for Parallel Computers CPC'96.* , pages 245–259, Aachen - Germany, December 1996.

[ES96b] Christine Eisenbeis and Antoine Sawaya. Optimal Loop Parallelization under Register Constraints. Technical Report RR-2781, INRIA, January 1996. ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-2781.ps.gz.

[FL98] Martin Farach and Vincenzo Liberatore. On Local Register Allocation. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 564–573, San Francisco, California, January 1998. ACM Press.

[FM01] D. Fimmel and J. Muller. Optimal Software Pipelining Under Resource Constraints. *International Journal of Foundations of Computer Science (IJFCS)*, 12(6):697–718, 2001.

[GAG94] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing Register Requirements under Resource-Constrained Rate-Optimal Software Pipelining. In *MICRO27*, pages 85–94, December 1994.

[GQD94] Guang R. Gao, Ning Qi, and Vincent Van Dongen. Extending Software Pipelining Techniques for Scheduling Nested Loops. *Lecture Notes in Computer Science*, 768:340–??, 1994.

[GS94] Franco Gasperoni and Uwe Schwiegelshohn. Generating Close to Optimum Loop Schedules on Parallel Processors. *Parallel Processing Letters*, 4(4):391–403, December 1994.

[GT86] Z. Galil and E. Tardos. An $O(n^2(m + n \log n) \log n)$ Min-Cost Flow Algorithm. In *27th Annual Symposium on Foundations of Computer Science*, pages 1–9, Los Angeles, Ca., USA, October 1986. IEEE Computer Society Press.

[H+92] L. J. Hendren et al. Register Allocation Using Cyclic Interval Graphs: A New Approach to an Old Problem. ACAPS Technical Memo 33, Advanced Computer Architecture and Program Structures Group, McGill University, Montreal, Canada, 1992.

[HGAM92] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs. *Lecture Notes in Computer Science*, 641:176–??, 1992.

[Hua01] Guillaume Huard. *Algorithmique du Décalage d'Instructions*. PhD thesis, Ecole Normale Supérieure, Lyon, France, December 2001.

[Huf93] R. Huff. Lifetime-Sensitive Modulo Scheduling. In *PLDI 93*, pages 258–267, Albuquerque, New Mexico, June 1993.

[Jan01] Johan Janssen. *Compilers Strategies for Transport Triggered Architectures*. PhD thesis, Delft University, Netherlands, 2001.

[Kes98] Christoph W. Kessler. Scheduling Expression DAGs for Minimal Register Need. *Computer Languages*, 24(1):33–53, April 1998.

[KPF95] Steven M. Kurlander, Todd A. Proebsting, and Charles N. Fischer. Efficient Instruction Scheduling for Delayed-Load Architectures. *ACM Transactions on Programming Languages and Systems*, 17(5):740–776, September 1995.

[Lan74] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. Chelsea Publishing Company, 1974. Reprinted from the First Edition, 1909.

[Law72] E. L. Lawler. Optimal Cycles on Graphs and Minimal Cost-to-Time Ratio Problem. In A. Marzojlo, editor, *Periodic Optimization*, volume 1, pages 38–58. Springer-Verlag, 1972.

[LB94] David J. Lilja and Peteto L. Bird. *The Interaction of Compilation Technology and Computer Architecture.* Kluwer Academic, Boston/Dordrecht/London, 1994.

[Lel96] Sylvain Lelait. *Contribution à l'Allocation de Registres dans les Boucles.* PhD thesis, Université d'Orléans, France, January 1996.

[LGAV96] J. Llosa, A. Gonzalez, E. Ayguadé, and M. Valero. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In *PACT 96*, Boston, Massachusetts, October 20-23 1996.

[Llo96] Josep Llosa. *Reducing the Impact of Register Pressure on Software Pipelined Loops.* PhD thesis, Universitat Politecnica de Catalunya (Spain), 1996.

[LMEG96] J. Llosaand, M.Valero, E.Ayguandé, and A. González. Heuristics for Register-Constrained Software Pipelining. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 250–261, Paris, France, December 1996.

[LS91] Charles E. Leiserson and James B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6:5–35, 1991.

[LVA95] J. Llosa, M. Valero, and E. Ayguadé. Hypernode Reduction Modulo Scheduling. In *micro28*, pages 350–360, Boston, Massachusetts, November 1995.

[MD99] Waleed M. Meleis and Edward S. Davidson. Dual-Issue Scheduling with Spills for Binary Trees. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 678–686, New York, January 17–19 1999. ACM-SIAM.

[Mel01] Waleed M. Meleis. Dural-Issue Scheduling for Binary Trees with Spills and Pipelined Loads. *SIAM J. Comput.*, 30(6):1921–1941, March 2001.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform of Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, England, January 1999.

[MSAD92] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson. Register Requirements of Pipelined Processors. In ACM, editor, *Conference proceedings / 1992 International Conference on Supercomputing, July 19–23, 1992, Washington, DC*, pages 260–271, New York, NY 10036, USA, 1992. ACM Press.

[Nak67] I. Nakata. On Compiling Algorithms for Arithmetic Expressions. *Communications of the ACM*, 10:492–494, July 1967.

[NG93] Qi Ning and Guang R. Gao. A Novel Framework of Register Allocation for Software Pipelining. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–42, Charleston, South Carolina, January 1993. ACM Press.

[Nin93] Qi Ning. *Optimal Register Allocation to Support Time Optimal Scheduling for Loops.* PhD thesis, School of Computer Science, McGill University, Montreal, Quebec, Canada, 1993.

[Orl88] James Orlin. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. In Richard Cole, editor, *Proceedings of the 20th Annual ACM Symposium on the Theory of Computing*, pages 377–387, Chicago, May 1988. ACM Press.

[PF91] Todd A. Proebsting and Charles N. Fischer. Linear-Time, Optimal Code Scheduling for Delayed-Load Architectures. *SIGPLAN Notices*, 26(6):256–267, June 1991. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.

[Ram94] J. Ramanujam. Optimal Software Pipelining of Nested Loops. In Howard Jay Siegel, editor, *Proceedings of the 8th International Symposium on Parallel Processing*, pages 335–343, Los Alamitos, CA, USA, April 1994. IEEE Computer Society Press.

[Red69] R. R. Redziejowski. On Arithmetic Expressions and Trees. *Communications of the ACM*, 12(2):81–84, February 1969.

[RLTS92] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register Allocation for Software Pipelined Loops. *SIGPLAN Notices*, 27(7):283–299, July 1992. Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.

[Saw97] Antoine Sawaya. *Pipeline Logiciel: Découplage et Contraintes de Registres.* PhD thesis, Université de Versailles Saint-Quentin-En-Yvelines, April 1997.

[SC96] Fermin Sanchez and Jordi Cortadella. RESIS: A New Methodology for Register Optimization in Software Pipelining. In *Proceedings of Second International Euro-Par Conference, Euro-Par'96*, Lyon, France, August 1996.

[SCFS98] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-Independent Storage Mapping for Loops. *ACM SIG-PLAN Notices*, 33(11):24–33, November 1998.

[Sch87] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, New York, 1987.

[Set75] R. Sethi. Complete register allocation problems. *SIAM Journal on Computing*, 4(3):226–248, 1975.

[SRM94] Schlansker, B. Rau, and S. Mahlke. Achieving High Levels of instruction-Level Parallelism with Reduced Hardware Complexity. Technical Report HPL-96-120, Hewlet Packard, 1994.

[SU70] R. Sethi and J. D. Ullman. The Generation of Optimal Code for Arithmetic Expressions. *Journal of the ACM*, 17(4):715–728, 1970.

[TE01] Sid-Ahmed-Ali Touati and Christine Eisenbeis. Schedule Independent Register Allocation for Software Pipelining. In *9th Workshop on Compilers for Parallel Computers*, Edinburgh, Scotland, UK, June 2001. ftp.inria.fr/INRIA/Projects/a3/touati/touati_cpc01.ps.gz.

[TE02] Sid-Ahmed-Ali Touati and Christine Eisenbeis. Cyclic Register Pressure and Allocation for Modulo Scheduled Loops. Research Report, INRIA, December 2002. ftp.inria.fr/INRIA/Projects/a3/touati/SIRA.ps.gz.

[Tou01a] Sid-Ahmed-Ali Touati. EquiMax: A New Formulation of Acyclic Scheduling Problem for ILP Processors. In *Interaction between Compilers and Computer Architectures*. Kluwer Academic Publishers, 2001.

[Tou01b] Sid-Ahmed-Ali Touati. Maximizing for Reducing Register Need in Acyclic Schedules. In *Proceedings of 5th International Workshop on Software and Compilers for Embedded Systems, SCOPES*, St Goar, Germany, March 2001.

[Tou01c] Sid-Ahmed-Ali Touati. Optimal Acyclic Fine-Grain Schedule with Cache Effects for Embedded and Real Time Systems. In *Proceedings of 9th nternational Symposium on Hardware/Software Codesign, CODES*, Copenhagen, Denmark, April 2001. ACM.

[Tou01d] Sid-Ahmed-Ali Touati. Optimal Register Saturation in Acyclic Superscalar and VLIW Codes. Research Report RR-4263, INRIA, September 2001. ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-4263.ps.gz.

[Tou01e] Sid-Ahmed-Ali Touati. Register Saturation in Superscalar and VLIW Codes. In *Proceedings of The International Conference on Compiler Construction*, Lecture Notes in Computer Science. Springer-Verlag, April 2001.

[TT00] Sid-Ahmed-Ali Touati and François Thomasset. Register Saturation in Data Dependence Graphs. Research Report RR-3978, INRIA, July 2000. ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-3978.ps.gz.

[Tuc75] Alan Tucker. Coloring a Family of Circular Arcs. *SIAM Journal on Applied Mathematics*, 29(3):493–502, November 1975.

[TVSA01] William Thies, Frederic Vivien, Jeffrey Sheldon, and Saman Amarasinghe. A Unified Framework for Schedule and Storage Optimization. *ACM SIGPLAN Notices*, 36(5):232–242, May 2001.

[WEJS94] Jian Wang, Christine Eisenbeis, Martin Jourdan, and Bogong Su. DEcomposed Software Pipelining: A new perspective and a new approach. *International Journal of Parallel Programming*, 22(3):351–373, June 1994.

[WKE95] Jian Wang, Andreas Krall, and M. Anton Ertl. Decomposed Software Pipelining with Reduced Register Requirement. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT95*, pages 277 – 280, Limassol, Cyprus, June 1995.

[WKEE94] Jian Wang, Andreas Krall, M. Anton Ertl, and Christine Eisenbeis. Software Pipelining with Register Allocation and Spilling. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 95–99, San Jose, California, November 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

[ZW01] T. Zeitlhofer and B. Wess. Integrated Scheduling and Register Assignment for VLIW-DSP Architectures. In *Proceedings of the 14th IEEE International ASIC/SOC Conference*, Washington DC, USA, September 2001.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399